Bertec Device Interface Library

SDK and API Documentation

Version 2.20
May 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 Library 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 but are not limited to:

1. BertecDevice.dll
2. 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 Windows 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, the Bertec Device Library 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 is presented to the consumer application by either on-demand polling or by various callback mechanisms. Both 32 bit and 64 bit Windows operating systems and applications are supported, and both standard “C” and .NET interfaces are provided.

Sample code is provided in the BertecExample.cpp (C/C++) and BertecExample.cs (C#) files. This document covers the C/C++ interface specifically; a separate document covers the C# interface.

If you have any developmental questions on using this library or example code, 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 consists of a DLL file (BertecDevice.DLL) that exposes all of the functionality that should be deployed along with your application, and a header (bertecif.h) file that will need to be included in whatever manner your development environment suggests. The DLL comes with a BertecDevice.LIB file that will need to be linked against your application – this DLL can be linked either statically or dynamically, and has not other dependencies other than the FTD2XX.DLL file.

In order for the Library to function 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 the FTD2XX.DLL file is not accessible, the `bertec_Init` function will return a NULL handle value and the Windows API function `GetLastError` will return `ERROR_FILE_NOT_FOUND`.

GATHERING DATA

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

1. Call `bertec_Init` to load the device driver DLL and get a handle to the Library.
2. Call `bertec_RegisterDataCallback` if your application supports threaded callbacks.
3. Call `bertec_Start`.
4. Wait for connected devices to become ready by using either `bertec_GetStatus` or a Status Callback.
5. Poll using `bertec_ReadBufferedData`, or use the registered threaded callback.
6. Perform any operation your application needs, such as data collection or analysis.
7. Call `bertec_Stop`.
8. End by calling `bertec_Close`.

Step 1: `bertec_Init`.

Calling `bertec_Init` will set up the internal data in the Library and ready it for use; you must do this before you can use any other function call in the Bertec Device Library. The `bertec_Init` function returns a handle that you must store and use later to pass to the other `bertec_XXX` functions; you should also check if this handle value is NULL to determine if the FTD2XX.DLL file was able to be loaded.

Step 2: `bertec_RegisterDataCallback`.

Depending on how your application works, you will either want to poll for the data yourself (pull) and process it, or else use the faster callback functionality (push). If your application uses callbacks (the suggested method), you will need to register the callback with the Library prior to calling `bertec_Start`.

Step 3: `bertec_Start`.

To actually detect any connected devices and begin gathering data, you must call `bertec_Start` with the handle that `bertec_Init` returned to you. 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 setup a data callback via `bertec_RegisterDataCallback`, then data will be presented to that function; otherwise your application need to repeatedly call `bertec_ReadBufferedData` in order to retrieve any buffered data.

Step 4: Wait for connected devices to become ready.

By using either the `bertec_GetStatus` function or else a registered `bertec_StatusCallback` 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 `BERTEC_DEVICES_READY` you can expect to start getting data on either the callback or the polling function.

Step 5: Handle data via a `bertec_DataCallback` function or else poll using `bertec_ReadBufferedData`

If your application registered a data callback, it will get 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 `bertec_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: `bertec_Stop`.

Once you have completed your data gathering call `bertec_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 `bertec_ReadBufferedData` will return an empty result. To resume data collection your application must call `bertec_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 `bertec_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: `bertec_Close`.

Once you are completely done with the Library, you will need to call `bertec_Close` 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 `bertec_ReadBufferedData` function until it returns a value indicating there is no more data left in the internal buffer. A helper function `bertec_GetBufferedDataAvailable` 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 `bertec_GetBufferedDataAvailable` 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 samples are discarded unless `bertec_ChangeMaxBufferedDataSize` is called 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:

```
bertec_Handle handle = bertec_Init();
bertec_Start(handle);
while (bertec_GetStatus(handle) != BERTEC_DEVICES_READY)
```

```
{
   waitingForDevices();
}
bertec_DataFrame data;
while (External_Flag_To_Keep_Running_Is_True)
{
   while (bertec_ReadBufferedData(handle,&data,sizeof(data)) > 0)
      processYourData(&data);
}
bertec_Stop(handle);
bertec_Close(handle);
```

Once the Library detects devices and performs the needed setup functions, the `bertec_GetStatus` function will return that the devices are ready. Your application would then proceed to the data collection loop with an allocated data frame buffer.

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 CALLBACKS

Callbacks are the preferred method for reading data from the Library. All callbacks 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 callbacks 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 callbacks, register your callback function along with an optional user data value with `bertec_RegisterDataCallback`. Whenever the callback is invoked, your function will receive a pointer to the dataframe along with the user data value that you set. Only one callback can be registered at a time; if your application needs to support more than one callback at a time you will need to implement some sort of list management system.

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

```
void myDataCallback(bertec_Handle hand, bertec_DataFrame * data, void * user)
{
  processYourData(data);
}

bertec_Handle handle = bertec_Init();
bertec_RegisterDataCallback(handle, myDataCallback, NULL);
bertec_Start(handle);

..your main program runs...

bertec_Stop(handle);
bertec_Close(handle);
```

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

ERROR CHECKING AND HANDLING

Most Library functions 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 defined in the header file.

Whenever the status of the Library changes, such as an error or loss of synchronization, any previously declared status callbacks set by `bertec_RegisterStatusCallback` are invoked with the new status value. The current status value can always be retrieved by calling the `bertec_GetStatus` function.

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 `BERTEC_DATA_BUFFER_OVERFLOW` error status will occur.

The data block (`bertec_DataFrame`) presented to your application will always be the same size, regardless of the total number of devices present on the system or the number of data channels each device has to offer. 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. The `bertec_DataFrame` structure is defined in the bertecif.h header file.

BERTEC DEVICE LIBRARY FUNCTIONS

All of the functions are “C” exported function calls. For each function call, the name is called out, and then the function call definition is provided, along with any relevant documentation and usage notes. A separate document covers the equivalent .NET interfaces.

BERTEC_LIBRARYVERSION

`unsigned int bertec_LibraryVersion (void)`

This function returns the current version of the library, which should always match the defined value in `BERTEC_LIBRARY_VERSION`. If it does not then it is highly likely that data structures and library functions have been changed and you should proceed with caution.

BERTEC_INIT

`bertec_Handle bertec_Init(void)`

The `bertec_Init` function initializes the Library and prepares it for use but does not start the actual device interaction – `bertec_Start` must be used to begin the data discovery and data collection process. Your must first call this function before using any other method (including registering any callback functions). The returned value is a `bertec_Handle` object that should be retained by your application to be used for future interface calls. Calling this multiple times is not recommended as it will force all existing connections closed and re-inits the Library.

If `bertec_Init` returns a NULL handle value, this indicates that the Library was unable to properly locate and load the FTD2XX.DLL file (the Windows function `GetLastError` will return a `ERROR_FILE_NOT_FOUND` value). This 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.

BERTEC_CLOSE

`void bertec_Close(bertec_Handle theHandle)`

Call the `bertec_Close` to shut down all devices, unregister all callbacks, and stops all data collection. You must pass it the same handle that `bertec_Init` provided. Failing to call this before unloading the Library or exiting your application may leave devices running and possibly introduce memory leaks. Calling this multiple times will have no effect.

BERTEC_CHECKHANDLE

`bool bertec_CheckHandle(bertec_Handle theHandle)`

Verifies that `theHandle` value passed is a valid `bertec_Handle` item. Returns FALSE if this is not so, and TRUE if the handle is valid. Provided so applications can validate their own runtime status and detect possible issues.

BERTEC_START

`int bertec_Start(bertec_Handle theHandle)`

This function starts the data gathering process, invoking callbacks 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 a `BERTEC_ERROR_INVALIDHANDLE` return code.

BERTEC_STOP

`int bertec_Stop(bertec_Handle theHandle)`

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

BERTEC_DATACALLBACK TYPEDEF

BERTEC_REGISTERDATACALLBACK

BERTEC_UNREGISTERDATACALLBACK

`void bertec_DataCallback(bertec_Handle theHandle, bertec_DataFrame * data, void * userData)`

`int bertec_RegisterDataCallback(bertec_Handle theHandle, bertec_DataCallback fn, void * userData)`

`int bertec_UnregisterDataCallback(bertec_Handle theHandle, bertec_DataCallback fn, void * userData)`

To use the data callback functionality as provided by the Library, you will need to register your callback function with `bertec_RegisterDataCallback`. Only one callback may be registered at a time; if your application needs to support multiple callbacks you will need to implement some sort of dispatching system. To stop using the callback without stopping data acquisition, call `bertec_UnregisterDataCallback`. Calling `bertec_Close` will automatically unregister all callbacks as will calling `bertec_Init`.

In order to properly unregister the callback, you must call `bertec_UnregisterDataCallback` with the same values as you registered it with – both the callback itself, and the value of `userData`. Failing to do so may not unregister the callback and return a negative result code.

Both the register and unregister functions return a zero value for success. You should register your callbacks before calling `bertec_Start` in order to insure that no data is lost, and the callback function should be a C-style function using the "standard call" specification. This is the default for most development environments.

Your callback function 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 `userData` parameter is set when your register the callback and is not used by the Bertec Device Library but is passed unchanged to your callback function. Typically, this is used as a pointer to a class or structure object. The `bertec_DataFrame` data pointer refers to an internal block of memory that is maintained by the Library and thus should not be deleted, freed, or otherwise modified by your application. The `userData` parameter is set when your register the callback, and is not used by the Bertec Device Library but may be used by your callback for whatever it needs – for example, in C++ it could be used to pass a pointer to a class object.

Since data collection and processing is time-critical, it is important that your applications processes the data block as quickly as possible and return.

Refer to the section on Data Frames for information about the format and type of the data block.

BERTEC_STATUSCALLBACK TYPEDEF

BERTEC_REGISTERSTATUSCALLBACK

BERTEC_UNREGISTERSTATUSCALLBACK

```
void bertec_StatusCallback(bertec_Handle theHandle, int status, void * userData)

int bertec_RegisterStatusCallback(bertec_Handle theHandle, bertec_StatusCallback fn, void *
userData)

int bertec_UnregisterStatusCallback(bertec_Handle theHandle, bertec_StatusCallback fn, void *
userData)
```

To use the status callback functionality you will need to register your callback function with `bertec_RegisterStatusCallback`. Only one callback may be registered at a time; if your application needs to support multiple callbacks you will need to implement some sort of dispatching system. To stop using the callback, call `bertec_UnregisterStatusCallback`. Calling `bertec_Close` will automatically unregister all callbacks.

In order to properly unregister the callback, you must call `bertec_UnregisterStatusCallback` with the same values as you registered it with – both the callback itself, and the value of `userData`. Failing to do so may not unregister the callback and return a negative result code.

Both the register and unregister functions return a zero value for success. You should register your callbacks before calling `bertec_Start` in order to insure that no data is lost, and the callback function should be a C-style function using the "standard call" specification. This is the default for most development environments.

Your callback function 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 calling `bertec_GetStatus` would return. These are defined in the header file and are also documented further down. The `userData` parameter is set when your register the callback, and is not used by the Bertec Device Library but may be used by your callback for whatever it needs – for example, in C++ it could be used to pass a pointer to a class object.

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

BERTEC_DEVICESORTCALLBACK TYPEDEF

BERTEC_REGISTERDEVICESORTCALLBACK

BERTEC_UNREGISTERDEVICESORTCALLBACK

```
void bertec_DeviceSortCallback (bertec_DeviceInfo* pInfos, int deviceCount, int* orderArray,
void * userData)

int bertec_RegisterDeviceSortCallback(bertec_Handle bHand, bertec_DeviceSortCallback fn, void *
userData)

int bertec_UnregisterDeviceSortCallback(bertec_Handle bHand, bertec_DeviceSortCallback fn, void
* userData)
```

To use the device sort order callback functionality you will need to register your callback function with `bertec_RegisterDeviceSortCallback`. To stop using the callback, call `bertec_UnregisterDeviceSortCallback`. Calling `bertec_Close` will automatically unregister all callbacks.

In order to properly unregister the callback, you must call `bertec_UnregisterDeviceSortCallback` with the same values as you registered it with – both the callback itself, and the value of `userData`. Failing to do so may not unregister the callback and return a negative result code.

Both functions return zero for success.

Your data callback function should be a C-style function, using the “standard call” specification. This is the default for most development environments.

The callback will be called each time the Library finishes discovering the list of devices attached to the computer. The `userData` parameter is set when your register the callback, and is not used by the Library but is to be used by your callback for whatever it needs – for example, in C++ it could be used to pass a pointer to a class object.

The `pInfos` pointer is the 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 `bertec_DeviceInfo` item in the `pInfos` 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.

Do not delete, free, or otherwise modify the `pInfos` array – it is maintained internally by the Library. You should not delete or free the `orderArray`, but it is expected that you change it to reflect your new device order.

BERTEC_DEVICETIMESTAMPCALLBACK TYPEDEF

BERTEC_REGISTERDEVICETIMESTAMPCALLBACK

BERTEC_UNREGISTERDEVICETIMESTAMPCALLBACK

```
int64_t bertec_DeviceTimestampCallback(int deviceNum, void * userData)
```

```
int bertec_RegisterDeviceTimestampCallback(bertec_Handle bHand, bertec_DeviceTimestampCallback
fn, void * userData)
```

```
int bertec_UnregisterDeviceTimestampCallback(bertec_Handle bHand, bertec_DeviceTimestampCallback
fn, void * userData)
```

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 time stamp callback functionality you will need to register your callback function with `bertec_RegisterDeviceTimestampCallback`. To stop using the callback, call `bertec_UnregisterDeviceTimestampCallback`. Calling `bertec_Close` will automatically unregister all callbacks.

In order to properly unregister the callback, you must call `bertec_UnregisterDeviceTimestampCallback` with the same values as you registered it with – both the callback itself, and the value of `userData`. Failing to do so may not unregister the callback and return a negative result code.

Both functions return zero for success.

Your data callback function should be a C-style function, using the “standard call” specification. This is the default for most development environments.

The callback 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 `bertec_AdditionalData.timestamp` data value.

The `userData` parameter is set when your register the callback, and is not used by the Library but is to be used by your callback for whatever it needs – for example, in C++ it could be used to pass a pointer to a class object. The example source code provided uses the `userData` value as the command line stepping parameter.

BERTEC_GETSTATUS

```
int bertec_GetStatus(bertec_Handle theHandle)
```

Returns the current status value of the Library, which will be the same value that would be passed to any `bertec_StatusCallback` that had been set. The status value will be one of the defined error numbers from the header file or zero, which indicates no error.

BERTEC_GETBUFFEREDDATAAVAILABLE

`int bertec_GetBufferedDataAvailable(bertec_Handle theHandle)`

This function 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 `bertec_ReadBufferedData` and should *not* be used when data callbacks have been set. This function 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 that there are at least that many data blocks available to be read, but there may be more.

BERTEC_READBUFFEREDDATA

`int bertec_ReadBufferedData(bertec_Handle bHand, bertec_DataFrame * dataFrame, size_t dataFrameSize)`

Instead of using the callbacks, you can use the `bertec_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 `bertec_ReadBufferedData` will take one data frame from the internal buffer, copying the values into your passed pointer 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.

This call will always return at most 1 data frame, never more. Your `dataFrameSize` parameter ***must*** be equal to or greater than the size of the outgoing data buffer, and account for the number of devices that are currently detected. Passing the wrong size will result in an `BERTEC_INVALID_PARAMETER` error. The needed size computation is:

```
dataFrameSize = sizeof( bertec_DeviceData ) * DeviceCount + sizeof( bertec_DataFrame );
```

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

```
while (bertec_ReadBufferedData(hand,buff,buffsize) > 0)
{
   processData(buff);
}
```

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

Using the polling function is not recommended; in most cases you will get better performance if you use the Data Callback feature and perform data buffering.

BERTEC_CLEARBUFFEREDDATA

`int bertec_ClearBufferedData(bertec_Handle bHand)`

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

BERTEC_GETMAXBUFFEREDDATASIZE

`int bertec_GetMaxBufferedDataSize(bertec_Handle bHand)`

Returns how many data samples that the Library will buffer before discarding old data. The default is 100 samples.

BERTEC_CHANGEMAXBUFFEREDDATASIZE

`int bertec_ChangeMaxBufferedDataSize(bertec_Handle bHand,int newMaxSamples)`

Changes the maximum amount of buffered data before the Library begins to discard old values. By default this 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 the `bertec_Start` function.

BERTEC_GETDEVICECOUNT

`int bertec_GetDeviceCount (bertec_Handle bHand)`

Returns the number of supported devices connected to the computer. Only valid once `bertec_Start` has been called and devices have actually been found (that is, `bertec_GetStatus` or the `bertec_StatusCallback` callback indicates a status value equal to `BERTEC_DEVICES_READY`).

BERTEC_GETDEVICEINFO

`int bertec_GetDeviceInfo(bertec_Handle bHand, int deviceIndex, bertec_DeviceInfo * info, size_t infoSize)`

Copies the information about the given device index into the info buffer and returns zero. If there is no device at the given index then this function will return a `BERTEC_INDEX_OUT_OF_RANGE` error and leave the `info` pointer contents untouched.

BERTEC_GETDEVICESERIALNUMBER

`int bertec_GetDeviceSerialNumber(bertec_Handle bHand, int deviceIndex, char *buffer, size_t bufferSize)`

This is a convenience function that will return a device's serial number directly instead of reading the entire device info into a `bertec_DeviceInfo` struct and accessing the `serial` member. If there is no device at the given index then this function will return a `BERTEC_INDEX_OUT_OF_RANGE` error and leave the `buffer` pointer contents untouched.

BERTEC_GETDEVICEMODELNUMBER

`int bertec_GetDeviceModelNumber(bertec_Handle bHand, int deviceIndex, char *buffer, size_t bufferSize)`

This is a convenience function that will return a device's model number directly instead of reading the entire device info into a `bertec_DeviceInfo` struct and accessing the `model` member. If there is no device at the given index then this function will return a `BERTEC_INDEX_OUT_OF_RANGE` error and leave the `buffer` pointer contents untouched.

BERTEC_GETDEVICECHANNELS

`int bertec_GetDeviceChannels(bertec_Handle bHand,int deviceIndex,char *buffer,size_t bufferSize)`

This is a convenience function that will return a copy of the channel names directly instead of reading the entire device info into a `bertec_DeviceInfo` struct and accessing the `channelNames` member. If there is no device at the given index then this function will return a `BERTEC_INDEX_OUT_OF_RANGE` error and leave the `buffer` pointer contents untouched. Otherwise it will return the number of channels for the device (ex: 3 for a device that has Fz, Mx, and My). The channel names are null-separated. For example, if the device has the channels Fz, Mx, and My, the buffer contents will contain the equivalent "C" string of `"FZ\0\MX\0\MY\0\0"`.

BERTEC_GETDEVICECHANNELNAME

`int bertec_GetDeviceChannelName(bertec_Handle bHand,int deviceIndex,int channelIndex,char *buffer,size_t bufferSize)`

This is a convenience function that will return a copy of a single channel name directly instead of reading the entire device info into a `bertec_DeviceInfo` struct and accessing the `channelNames` member. If there is no device at the given index then this function will return an `BERTEC_INDEX_OUT_OF_RANGE` error and leave the `buffer` pointer contents untouched. Returns the length of the channel name in ascii characters (ex: if device #0 has a channel named "FZ" at channel index #1, this will return a length of 2).

BERTEC_SETAVERAGING

`int bertec_SetAveraging(bertec_Handle bHand,int samplesToAverage)`

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

BERTEC_GETAVERAGING

`int bertec_GetAveraging(bertec_Handle bHand)`

Returns the currently set averaging value (by default 1) as set by `bertec_SetAveraging`.

BERTEC_SETLOWPASSFILTERING

`int bertec_SetLowpassFiltering(bertec_Handle bHand,int samplesToFilter)`

Performs a running average of the previous `samplesToFilter`, making the input data stream to appear smoother. The `samplesToFilter` value should be >= 2 in order to turn the filter on. Setting `samplesToFilter` to 1 or less will turn filtering off (the default). This does not affect the total number of samples gathered.

BERTEC_GETLOWPASSFILTERING

`int bertec_GetLowpassFiltering(bertec_Handle bHand)`

Returns the currently set low pass filtering value (by default 1) as set by `bertec_SetLowpassFiltering`.

BERTEC_ZERONOW

`int bertec_ZeroNow(bertec_Handle bHand)`

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 `bertec_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 `bertec_Start` will cause your data stream to rapidly change values as the new zero point is taken. This can be used in conjunction with `bertec_EnableAutozero`.

For best results you should call this right after your application first receives a `BERTEC_DEVICES_READY` value from a `bertec_GetStatus` call or a `bertec_StatusCallback` callback.

BERTEC_SETENABLEAUTOZERO

`int bertec_SetEnableAutozero(bertec_Handle bHand,int enableFlag)`

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). Calling this function with a non-zero value for `enableFlag` will cause the Library to monitor the data stream and continually reset the zero baseline values. Call this function with a zero value for `enableFlag` to turn this 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.

BERTEC_GETENABLEAUTOZERO

`int bertec_GetEnableAutozero(bertec_Handle bHand)`

Returns the currently set auto zero flag set by `bertec_SetEnableAutozero`.

BERTEC_GETAUTOZEROSTATE

`int bertec_GetAutozeroState(bertec_Handle bHand)`

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

The following values are returned from `bertec_GetAutozeroState` :

State	Value	Explanation
`AUTOZEROSTATE_NOTENABLED`	0	Autozeroing is currently not enabled.
`AUTOZEROSTATE_WORKING`	1	Autozero is currently looking for a sample to zero against.
`AUTOZEROSTATE_ZEROFOUND`	2	The zero level has been found and is being applied. The Library will continually attempt to zero automatically.

BERTEC_GETZEROLEVELNOISEVALUE

`double bertec_GetZeroLevelNoiseValue(bertec_Handle bHand,int deviceIndex,int channelIndex)`

Returns the zero level noise value for a device and channel. Either `bertec_ZeroNow` or `bertec_EnableAutozero` must have been called prior to this function being used. 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.

BERTEC_SETUSBTHREADPRIORITY

`void bertec_SetUsbThreadPriority(bertec_Handle bHand,int priority)`

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

BERTEC_SETENABLEMASTERSLAVEMODE

`int bertec_SetEnableMasterSlaveMode(bertec_Handle bHand,int enableMasterClock)`

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

If the ENABLE flag is TRUE 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. Calling this function with `enableMasterClock` set to FALSE after `bertec_Init` but before calling `bertec_Start` will prevent this and cause all devices to run as `SYNC_NONE`. You can change this at any point during runtime by calling this function with `enableMasterClock` set to the desired value.

Calling this with TRUE is exactly the same as calling `bertec_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.

BERTEC_GETENABLEMASTERSLAVEMODE

`int bertec_GetEnableMasterSlaveMode(bertec_Handle bHand)`

Returns the last set (or default) Master/Slave mode value. Only valid if there is more than one device currently connected.

BERTEC_SETSYNCPINMODE

`int bertec_SetSyncPinMode(bertec_Handle bHand,int deviceIndex,bertec_SyncModeFlags newMode)`

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 `bertec_DeviceData.additionalData.syncData` value.

`bertec_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.

BERTEC_SETAUXPINMODE

`int bertec_SetAuxPinMode(bertec_Handle bHand,int deviceIndex,bertec_AuxModeFlags newMode)`

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 `bertec_DeviceData.additionalData.auxData` value.

`bertec_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.

BERTEC_SETSYNCAUXPINVALUES

`int bertec_SetSyncAuxPinValues(bertec_Handle bHand,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.

BERTEC_RESETSYNCCOUNTERS

`int bertec_ResetSyncCounters(bertec_Handle bHand)`

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.

BERTEC_RESETDEVICECLOCK

`int bertec_ResetDeviceClock(bertec_Handle bHand, 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 `BERTEC_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.

BERTEC_RESETALLDEVICECLOCKS

`int bertec_ResetAllDeviceClocks(bertec_Handle bHand, 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 `BERTEC_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.

BERTEC_RESETDEVICECLOCKATTIME

`int bertec_ResetDeviceClockAtTime(bertec_Handle bHand, 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 possibly return `BERTEC_UNSUPPORED_COMMAND` error.

BERTEC_RESETALLDEVICECLOCKSATTIME

`int bertec_ResetDeviceClockAtTime(bertec_Handle bHand, 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 `BERTEC_UNSUPPORED_COMMAND` error.

BERTEC_SETEXTERNALCLOCKMODE

`int bertec_SetExternalClockMode(bertec_Handle bHand, int deviceIndex, bertec_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 `bertec_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.

BERTEC_SETAGGREGATEDEVICEMODE

BERTEC_GETAGGREGATEDEVICEMODE

`int bertec_SetAggregateDeviceMode(bertec_Handle bHand, int enable)`

`int bertec_GetAggregateDeviceMode(bertec_Handle bHand)`

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 output from the callback or data block will change accordingly. If this mode is turned on then the `bertec_DataFrame::deviceCount` value will be set to 1 even if there are more than one device connected, but `Bertec_GetDeviceCount` will always return the true number of devices connected to the system.

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.

BERTEC_SETCOMPUTEDCHANNELSFLAGS

BERTEC_GETCOMPUTEDCHANNELSFLAGS

`int bertec_SetComputedChannelsFlags(bertec_Handle bHand, bertec_ComputedChannelFlags newMode)`

`bertec_ComputedChannelFlags bertec_GetComputedChannelsFlags(bertec_Handle bHand)`

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 `bertec_SetSubjectHeight`. 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 `bertec_Init` but before `bertec_Start`; calling this while devices are actively delivering data will result in an error.

Setting these flags will affect both the channel names (`bertec_DeviceInfo::channelNames`, `bertec_GetDeviceChannels`, `bertec_GetDeviceChannelName`) and the actual data frame data (the channel count field in `bertec_DeviceData::channelData`)

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.

BERTEC_SETSUBJECTHEIGHT

`int bertec_SetSubjectHeight(bertec_Handle bHand, float heightMM)`

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.

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

BERTEC_DEVICEDATARATE

`float bertec_DeviceDataRate(bertec_Handle bHand)`

This function 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.

BERTEC_REDETECTCONNECTEDDEVICES

`int bertec_RedetectConnectedDevices(bertec_Handle bHand)`

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 BERTEC_LOOKING_FOR_DEVICES will be emitted via the Status callback (if any), followed by either BERTEC_NO_DEVICES_FOUND if there are no devices connected, or BERTEC_DEVICES_READY if one or more devices have been found.

This function does not block and returns immediately.

BERTEC_SETDEVICELOGDIRECTORY

`void bertec_SetDeviceLogDirectory(const char *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 function should be called prior to any other Library function, including `bertec_Init`; otherwise there will be log data split between two separate files as the directory changes.

BERTEC_GETCURRENTDEVICELOGFILENAME

`int bertec_GetCurrentDeviceLogFilename(char* buffer,int maxBufferSize)`

Fills the string buffer pointed to by the `buffer` parameter with the current log filename, including the path prefix, and returns the length of the string. Passing either NULL for the pointer or 0 for `maxBufferSize` will not fill the buffer but instead return how much space is needed.

This can be used to copy or otherwise reference the output logfile that is being created by the internal diagnostics of the Library.

BERTEC_DEVICELOGCALLBACK TYPEDEF

BERTEC_REGISTERDEVICELOGCALLBACK

BERTEC_UNREGISTERDEVICELOGCALLBACK

```
void bertec_DeviceLogCallback(const char* pszText, void * userData)
```

```
int bertec_RegisterDeviceLogCallback(bertec_Handle bHand, bertec_DeviceLogCallback fn, void *
userData)
```

```
int bertec_UnregisterDeviceLogCallback(bertec_Handle bHand, bertec_DeviceLogCallback fn, void *
userData)
```

This will set a callback that is invoked whenever a new line of text is being written to the device log file. The callback 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.

ERROR/STATUS CODES

Error	Value	Explanation
BERTEC_NO_BUFFERS_SET	-2	There are no internal buffers allocated. This is a critical error.
BERTEC_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.
BERTEC_NO_DEVICES_FOUND	-5	There are apparently no devices attached. Attach a device.
BERTEC_DATA_READ_NOT_STARTED	-6	You have not called `bertec_Start` yet.
BERTEC_DATA_SYNCHRONIZING	-7	Synchronizing, data not available yet.
BERTEC_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.
BERTEC_DATA_SEQUENCE_MISSED	-9	One or more plates have missing data sequence - data may be invalid
BERTEC_DATA_SEQUENCE_REGAINED	-10	The plates have regained their data sequence
BERTEC_NO_DATA_RECEIVED	-11	No data is being received from the devices, check the cables
BERTEC_DEVICE_HAS_FAULTED	-12	The device has failed in some manner. Power off the device, check all connections, power back on
BERTEC_LOOKING_FOR_DEVICES	-45	Scanning for any connected devices. This status will be followed by either BERTEC_DATA_DEVICES_READY or BERTEC_NO_DEVICES_FOUND.
BERTEC_DATA_DEVICES_READY	-50	There are plates to be read
BERTEC_AUTOZEROSTATE_WORKING	-51	Currently finding the zero values

BERTEC_AUTOZEROSTATE_ZEROFOUND	-52	The zero leveling value was found
BERTEC_ERROR_INVALIDHANDLE	-100	The `bertec_Handle` passed to a function is incorrect.
BERTEC_UNABLE_TO_LOCK_MUTEX	-101	Unable to properly manage a thread mutex context. This is a fatal error.
BERTEC_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.
BERTEC_INVALID_PARAMETER	-201	One or more of the parameters to a function call are incorrect (ex: null pointer, negative size, etc).
BERTEC_INDEX_OUT_OF_RANGE	-202	The device index value is negative or more than the number of devices currently attached to the system.
BERTEC_GENERIC_ERROR	-32767	Any error that has no predefined value.

TROUBLESHOOTING

If your application will not launch, make sure that both the BertecDevice.dll and ftd2xx.dll are in the same folder as your application.

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
03/24/2014	1.82	Added sort ordering, usb thread priority functions, 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
5/23/2019	2.20	Added the Check Handle function and additional verbiage around the Polling function. Expanded the number of concurrent devices from 4 to 32.	Todd Wilson