Bertec Device Interface Library

SDK and API Documentation

Version 2.50
August 2022

Printed in the United States of America.

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, 2500 Citygate Drive, Columbus, Ohio 43219

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, data synchronization (both external and internal), and synthetic computed data channels. 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).

Frame Rate: The rate at which the *hardware* device samples and transmits data to over the USB connection. The frame rate is generally independent between devices and runs at fixed a 1kHz (1000Hz) rate. This is exposed in the Data Frame as the 'timestamp' value.

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 and has no other dependencies outside of the required Future Technology Devices FTD2XX.DLL file.

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_RegisterDataStreamCallback` 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. Set the desired Data Stream mode using `bertec_StartDataStream`.
6. Poll using `bertec_ReadBufferedDataStream`, or use the registered threaded callback.
7. Perform any operation your application needs, such as data collection or analysis.
8. Call `bertec_Stop`.
9. 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_RegisterDataStreamCallback`.

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_RegisterDataStreamCallback`, then data will be presented to that function; otherwise your application need to repeatedly call `bertec_ReadBufferedDataStream` 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: Set the desired Data Stream mode using `bertec_StartDataStream`.

Multiple Bertec devices can be used in various ways, both solo and combined with other hardware. For example, a 6800 amplifier can be connected to an external data signal to control how the data is delivered, either as a user-controllable clock or on-off switch. Multiple amplifiers can be bridged together to provided synchronized data.

`bertec_StartDataStream` provides a supported method for the Library to control this, only starting to deliver data once various selectable conditions are met. If `bertec_StartDataStream` is not called, then no data will be presented to the end point for consumption.

Classical non-synchronized mode can be set up by calling `bertec_StartDataStream` with an empty `bertec_DataStreamControl`, which will result in the data stream effectively using `SYNCPINMODE_NONE` and `AUXPINMODE_NONE`. See the `bertec_StartDataStream` section for more information.

Step 6: Handle data via a `bertec_DataStreamCallback` function or else poll using `bertec_ReadBufferedDataStream`

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_ReadBufferedDataStream` to retrieve the currently buffered data one block at a time.

Step 7: 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 8: `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_ReadBufferedDataStream` 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 9: `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_ReadBufferedDataStream` function until it returns a value indicating there is no more data left in the internal buffer. A related function `bertec_GetBufferedDataAvailable` can be used to pre-determine how much data is currently available in the internal buffer. Note that there may 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_StartDataStream( handle, NULL );
size_t datasize = 0;
bertec_DataFrame* data=bertec_AllocateReadBufferedData(handle, &datasize);
while (External_Flag_To_Keep_Running_Is_True)
{
   while (bertec_ReadBufferedDataStream(handle,data,datasize) > 0)
      processYourData(data);
}
bertec_FreeAllocatedReadBufferedData handle, 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 then starts the desired data stream mode (in this case, classical non-sync), and proceeds into a 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 until it has been set to false.

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. Memory allocation manager that is required for the data polling is also removed. 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_RegisterDataStreamCallback`. Whenever the callback is invoked, your function will receive a pointer to at `bertec_DataFrame` structure 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);
}

void myStausCallback(bertec_Handle hand, int status, void * user)
{
  if (status == BERTEC_DEVICES_READY)
  { // start the data stream
     bertec_StartDataStream( handle, NULL );
  }
}

bertec_Handle handle = bertec_Init();
bertec_RegisterDataStreamCallback(handle, myDataCallback, NULL);
bertec_RegisterStatusCallback(handle, myStausCallback, NULL);
bertec_Start(handle);

...your main program runs; the status callback starts the data streaming...

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

As opposed 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. You also do not need to create a memory buffer for the callback to place data into; memory is managed by the Library and your callback will get a pointer to a memory block.

DATA STREAMING AND MULTIPLE DEVICES

One of the main concerns with multiple devices is making sure the incoming data is temporally aligned. With external amplifiers such as the AM6500 and AM6800, this can be accomplished using a SYNC wire connecting all the amplifier's SYNC pins together. This provides a common reference signal that is shared between the devices. The Library will use this SYNC signal to govern the data synchronization and "reclocking" of the data rate. The source of the SYNC signal can be either internal (the amplifier generates the pulse train) or external (such as a frequency generator or some other hardware).

Note that stand-alone USB plates with no external amplifiers do not have a SYNC pin and thus cannot use the SYNC signal; for these devices, a 'best effort' is made to align the device data based on hardware clocks.

When synchronizing multiple devices, there are three broad modes:

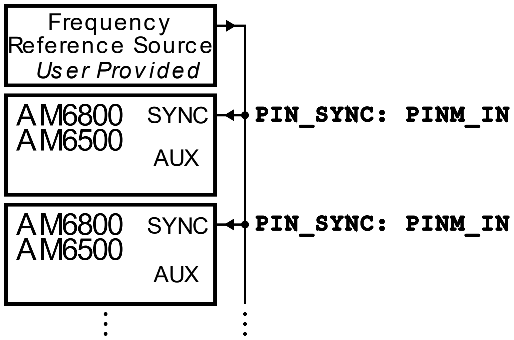
None: all the device's data is read and presented to the endpoint consumer (your `bertec_DataStreamCallback` method or the `bertec_ReadBufferedDataStream` function) once each device has presented a complete block of data. The data may not be temporally aligned and is the only mode available for stand-alone USB plates that do not have SYNC pins. This mode can also be used when direct control over the SYNC and AUX pins are desired.

Classic: a legacy mode provided for old hardware and firmware. In this mode, a master clock reference is provided by one device on the SYNC pin and the other devices use it. In this mode it is not possible to start or stop the data via the SYNC pin or resample the incoming data.

Clocked: the SYNC pin is used to receive an external clock signal (either from another amplifier or a secondary device), and the Library aligns the data on the starting edge of the signal. Data is resampled by the Library to match the frequency on the SYNC pin – thus with this mode, the apparent data rate can be varied from the fixed hardware rate of 1000Hz from a low as 150Hz to as high as 4000Hz. Data can be stopped and restarted mid-stream by simply pausing the clock signals on the SYNC pin. This mode also allows for a higher degree of data loss detection and recovery.

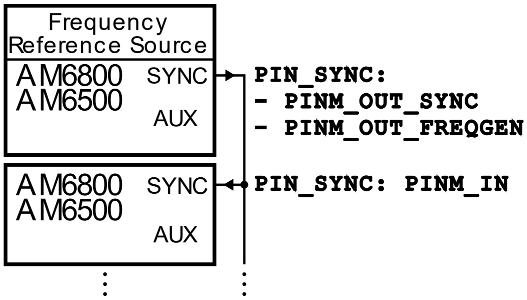
Using the Clocked method (passing either INTCLOCK or EXTCLOCK to the `bertec_DataStreamControl` structure) requires specific hardware setups and current amplifiers with current firmware. Using either method allowed, and the hardware setup is the similar with the only variation being the presence or lack of an external clock signal source.

For setups with an *external* clock source, a typical hardware setup would look like this:



Here the clock source is coming from the “User Provided” device.

For setups with an *internal* clock source (clock signal provided by a AM6500/AM6800), a typical hardware setup would look like this:



Here the clock source is coming from the AM6800/AM6500 device at the ‘top’ of the logical stack.

In both cases, `bertec_StartDataStream` with the “Clocking” mode will perform the following steps, once all devices are connected and detected (`bertec_GetStatus` returns DEVICES_READY):

1) Do all devices support the desired mode? If not, then `bertec_StartDataStream` will fail and return an error code. In this case, your application may wish to defer back to either “None” or “Classic”.
2) When *external* clocking is being used, there *must* be a “quiet period” of around 600ms that occurs during the setup phase of `bertec_StartDataStream` when no signals are being presented on the line (the line is logically held low or

neutral); for *internal* clocking the Library controls the SYNC line by setting it to output mode and setting the value to 0. This "quiet period" is used to validate the SYNC line connection and perform initial data frame alignments using the hardware's `bertec_AdditionalData::eventCounter` value.

3) If *internal* clocking is desired, the device selected in the data stream control block is set to generate pulses at the desired frequency.
4) Data will start to appear on your `bertec_DataStreamCallback` function (if used) and the `bertec_ReadBufferedDataStream` method *as long as clock signals are present on the SYNC line*.

With *internal* clocking (AM6500/AM6800 providing the clock), data will instantly begin to appear to the end point code (callback/polling). With *external* clocking, this data will NOT begin to appear UNTIL the clock source begins to generate clock signals. This can be either some form of hardware control (a button), a software signal to a 3rd part device, or some other method. Due to the way the SYNC pin is used to resample the data, stopping the external clock has the same effect as 'pausing' the data stream.

The external clock rate can vary from any value from approximately 150Hz to 4000Hz, inclusive. This variation can occur *at any time*, so a variable clock source does *not* require stopping and restarting the data stream.

IMMEDIATE DATA STREAMS

The Library provides a method where your application can receive data the instant each USB connection receives it, disregarding the other devices. This data is per-device only and is not synchronized or resampled in any way; only zero offsets are applied, and no computed channel logic is performed.

The Immediate data appears the instant a device is connected and delivers data to the PC. This delivery occurs outside of the Data Stream logic and is generally provided as a method for your application to monitor the device for a UI display or some other advanced operation.

Setting up the Immediate data callback is performed by using `bertec_RegisterImmediateDeviceDataCallback` with your callback function. Your function will get a data frame along with information about the device delivering the data.

A simple example of using Immediate data would be something like this:

```
void myImmDataCallback (bertec_Handle hand, int index, const char* uid, const bertec_DeviceData
* data, void * user)
{
  processTheData(index, uid, data);  // will be called by multple threads
}

bertec_Handle handle = bertec_Init();
bertec_RegisterImmediateDeviceDataCallback (handle, myImmDataCallback, NULL);
bertec_Start(handle);

...start the data stream or anything else; your program runs..

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

The `processTheData` method would do whatever your code needs but should return as quickly as possible. As noted, the callback method will be called by multiple threads so your code needs to allow for that. The Library will attempt to ensure that your code is called in a thread-safe manner.

See `bertec_RegisterImmediateDeviceDataCallback` for more information on Immediate Data.

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 (BERTEC_DATAFRAME)

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`, defined in the bertecif.h header file) presented to your application by either the callback mechanism or the data polling function will contain at least one device's worth of data, up to as many devices as there are physically connected. Device data is stored in the `bertec_DataFrame::device` array – your application should iterate through the `bertec_DataFrame::deviceCount` value for each of the `bertec_DeviceData` structures.

The `bertec_DeviceData` structure consists of two member structures: `bertec_ChannelData`, which contains the actual data from the plate and any computed channel values, and `bertec_AdditionalData`, which contains the frame counter, timestamp value, and hardware-specific SYNC and AUX input channels and event counters.

`bertec_ChannelData` consists of a result count and a floating-point array of channel data values. Each `bertec_ChannelData` structure contains from 1 to 32 channel values (`BERTEC_MAX_CHANNELS`). Device channels such as Fz, Mx etc. are presented in the order they occur within the device itself, and if the computed channel option is enabled (Cop, Cog, Sway), these will appear after the hardware values. See `bertec_SetComputedChannelsFlags` for more information.

`bertec_AdditionalData` contains a 64-bit frame counter, timestamp value, and the hardware-specific SYNC and AUX input channels and event counters. The timestamp value comes from the hardware device (if supported) and indicates when the force data itself was *sampled* by the hardware not when the PC received the data; the frame counter value is when the data was placed into the outgoing buffer (after being processed and subject to various settings such as down/up sampling, average, etc.). Typically, both the hardware timestamp and frame counter will increment at the same 1kHz monatomic rate. Depending on the settings these values can diverge, and the hardware timestamp can skip or repeat; the frame counter will never skip nor repeat.

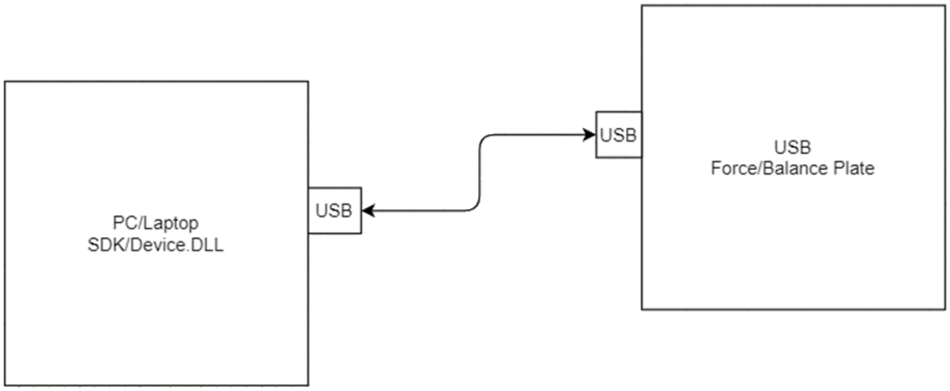
The hardware samples the force data at a 1kHz rate, while it samples the SYNC and AUX pin lines at an 8kHz rate (thus both the SYNC and AUX values are 8-bit LSB patterns, with a 1 bit value indicating the signal line was at a TTL +5v level, and 0 when it was at ground). If the hardware supports it, the event counter value indicates the in-hardware edge detection for either the SYNC or AUX values; see `bertec_SetExternalClockMode`, `bertec_SetSyncPinMode`, and `bertec_SetAuxPinMode`. These modes can be used to change the apparent data rate speed (external clocking) or can be used as an additional input control (example: a digital on/off switch as shown in BertecExample.cpp).

SPECIFIC USE CASES

Most of the time, simply calling Init, Start, and then StartDataStream will be all you need or want to do. However, there are specific use cases that might apply to your implementation so knowing about them ahead of time can be helpful.

SINGLE USB FORCE PLATE, NO SYNC

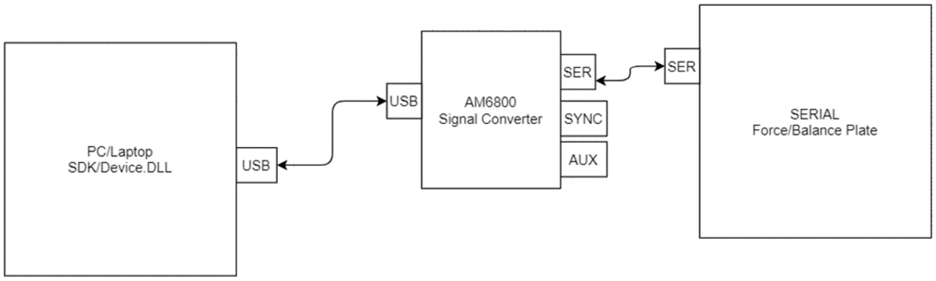
This is probably the most common mode of operation. A single force plate is connected via USB to the system, with no SYNC or AUX pins available. The data rate in this case is fixed at 1000Hz and cannot be changed via resampling.



Implementing this use case in your code is the same as the code shown in *Using Data Polling* and *Using Callbacks*. The sample file **BertecExample.cpp** provides a complete example suitable for command-line programs, and the sample file **SimpleFZReaderExample.cpp** provides a stripped-down version that picks out the Fz channel and displays only that.

SINGLE AMPLIFER, NO SYNC

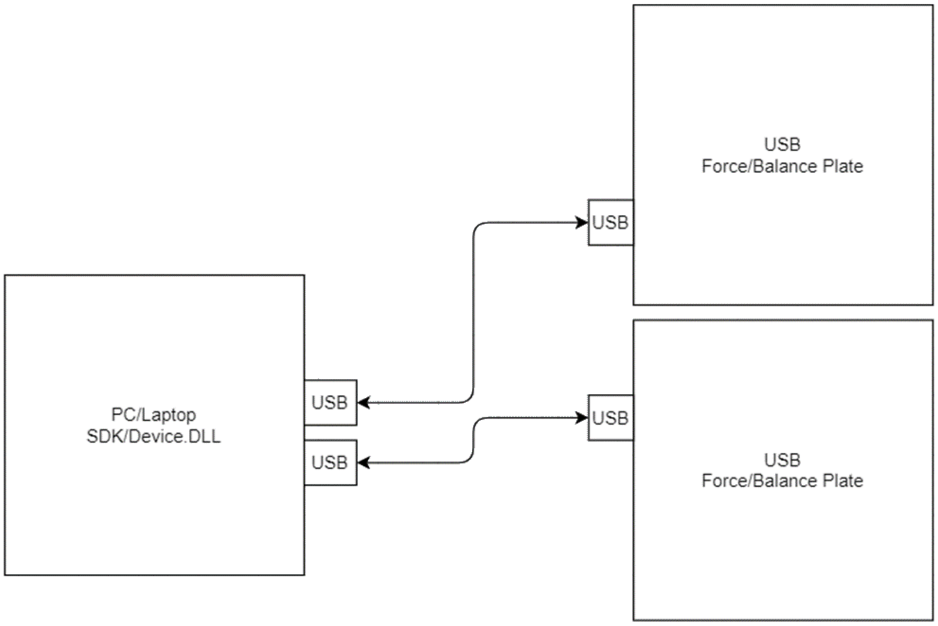
Along with the single USB plate, this is also a common mode of operation. A single force plate is connected to either an AM6500 or AM6800 amplifier, which is then connected to the PC via USB. While both the SYNC and AUX pins are available, they are not used. The data rate in this case is fixed at 1000Hz and cannot be changed via resampling.



Implementing this use case in your code is no different than *Single USB Force Plate No Sync* or *Using Data Polling* and *Using Callbacks*. The same sample code files **BertecExample.cpp** and **SimpleFZReaderExample.cpp** apply to this use case.

MULTIPLE USB FORCE PLATES, NO SYNC

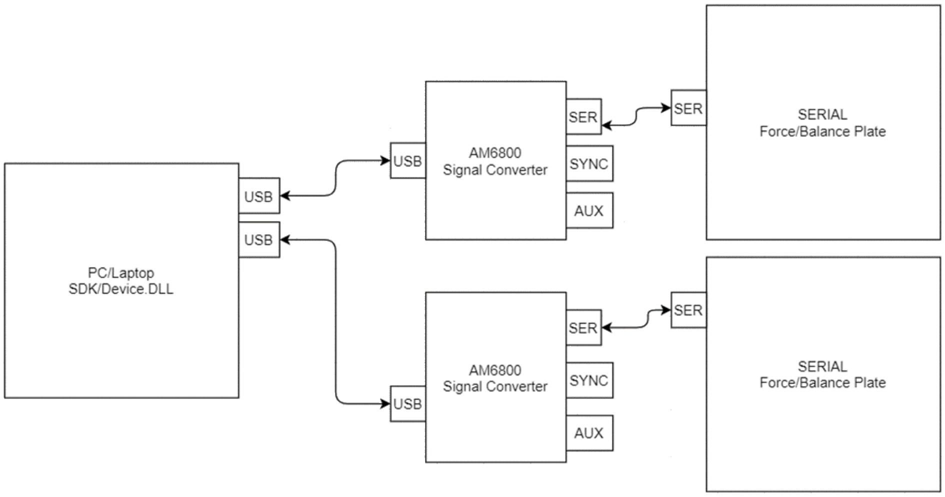
While less common than a single plate, multiple USB plates are often used in conjunction to provide a larger working area for the subject. The data rate is fixed at 1000Hz and cannot be changed via resampling.



Implementing this use case in your code requires no additional changes from *Single USB Force Plate No Sync* or *Using Data Polling* – the setup to `StartDataStream` remains the same, as does the data processing. The only difference between this use case and a single plate is that the `bertec_DataFrame::deviceCount` value will not be 1. Depending on the needs of your application, you can either loop through the `bertec_DataFrame::device[]` array or elect to ignore anything beyond the 1st device. Both sample files **BertecExample.cpp** and **SimpleFZReaderExample.cpp** loop through the device counts to output the force values.

MULTIPLE AMPLIFERS, NO SYNC

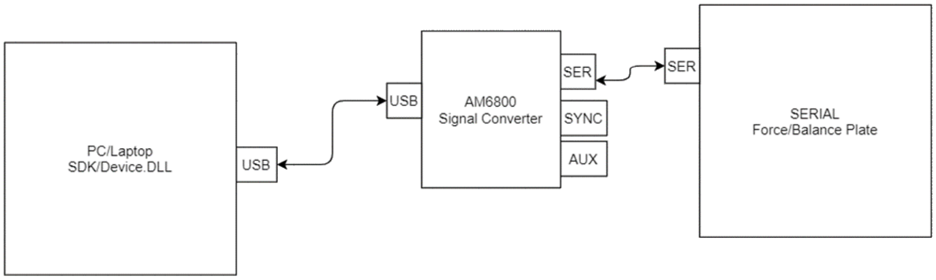
Multiple amplifiers present the same use case interface as multiple USB plates, and code-wise 'appear' the same except for the SYNC and AUX pins being available but not used. The data rate is fixed at 1000Hz and cannot be changed via resampling.



Implementing this use case in your code is no different than *Multiple USB Force Plates No Sync* or *Using Data Polling* and *Using Callbacks*. The same sample code files **BertecExample.cpp** and **SimpleFZReaderExample.cpp** apply to this use case, as does any device array handling your program implements.

INTERNAL CLOCK RESAMPLING, SINGLE PLATE

This use case is desirable for when your application requires control over the data rate. The Amplifier is set to read its own SYNC pin that is generating clock pulses, and those clock pulses are used to drive the resampling logic. The SYNC pin becomes an output, and while an external device can *read* it, it should not be injecting signals into it. Typically, nothing is connected to the SYNC pin for this use case.



For this mode, you would want to pass `bertec_DataStreamControl::SYNCPINMODE_INTCLOCK` to the `syncPinMode` field, along with a desired generation frequency (the changed lines from the generic case are highlighted). For example:

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

void myStausCallback(bertec_Handle hand, int status, void * user)
```

```
{
  if (status == BERTEC_DEVICES_READY)
  { // start the data stream
    bertec_DataStreamControl streamControl = { 0 };
    streamControl.size = sizeof( streamControl );
    streamControl.syncPinMode = bertec_DataStreamControl::SyncPinMode::SYNCPINMODE_INTCLOCK;
    streamControl.internalClockFrequency = 500; // 500 hz
    bertec_StartDataStream( handle, &streamControl );
  }
}

bertec_Handle handle = bertec_Init();
bertec_RegisterDataStreamCallback(handle, myDataCallback, NULL);
bertec_RegisterStatusCallback(handle, myStausCallback, NULL);
bertec_Start(handle);

...your main program runs; the status callback starts the data streaming...

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

This code will set up the device to generate 500Hz pulse signals on the SYNC pin, which the amplifier will read back into the data stream on the `syncData` value. The resampler logic inside the system will use this to ‘downsample’ the data into an apparent 500Hz data rate. The sample file **InternalClockingFZReaderExample.cpp** shows this in action.

You can change the rate after the fact by using the function `bertec_SetFrequencyGeneration` with the new value. For example:

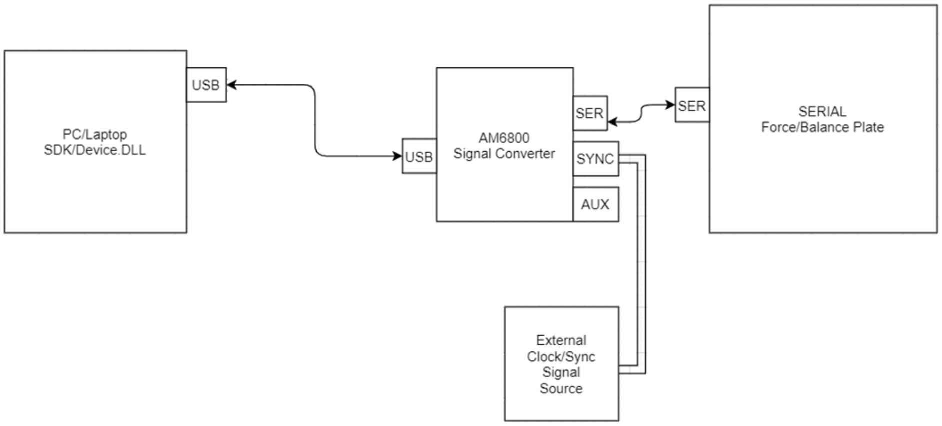
```
bertec_SetFrequencyGeneration( handle, 0, bertec_IOPins::IO_PIN_SYNC, 2000);
```

Will change the SYNC pin output and resulting resampling rate from the above 500Hz to 2000Hz. The data *frame rate* from the hardware is still fixed at 1000Hz, but the data is now being resampled into 2000Hz.

Special note: when using Internal Clock, you *must* set the `internalClockFrequency` value to a valid value (such as 250 or 1000); setting this to zero or a negative number will result in an error.

EXTERNAL CLOCK RESAMPLING, SINGLE PLATE

This use case is for when there is an external device providing clock signals to the amplifier on the SYNC pin, where the external clock controls both the timing of the data and the resampling of it. The external clock pulses are used to drive the resampling logic. The SYNC pin becomes an input. A typical use for such a setup would be cameras that provide a reference clock pulse.



For this use case, you would want to pass `bertec_DataStreamControl::SYNCPINMODE_EXTCLOCK` to the `syncPinMode` field instead of INTCLOCK. For example:

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

void myStausCallback(bertec_Handle hand, int status, void * user)
{
  if (status == BERTEC_DEVICES_READY)
  { // start the data stream
    bertec_DataStreamControl streamControl = { 0 };
    streamControl.size = sizeof( streamControl );
    streamControl.syncPinMode = bertec_DataStreamControl::SyncPinMode::SYNCPINMODE_EXTCLOCK;
    // Note there is no frequency being set
    bertec_StartDataStream( handle, &streamControl );
  }
}

bertec_Handle handle = bertec_Init();
bertec_RegisterDataStreamCallback(handle, myDataCallback, NULL);
bertec_RegisterStatusCallback(handle, myStausCallback, NULL);
bertec_Start(handle);

...your main program runs; the status callback starts the data streaming...

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

The data streamer will start to deliver data once the external clock device delivers clock pulses on the SYNC pin. For best results, the external clock *should not* be delivering pulses until *after* StartDataStream has been called.

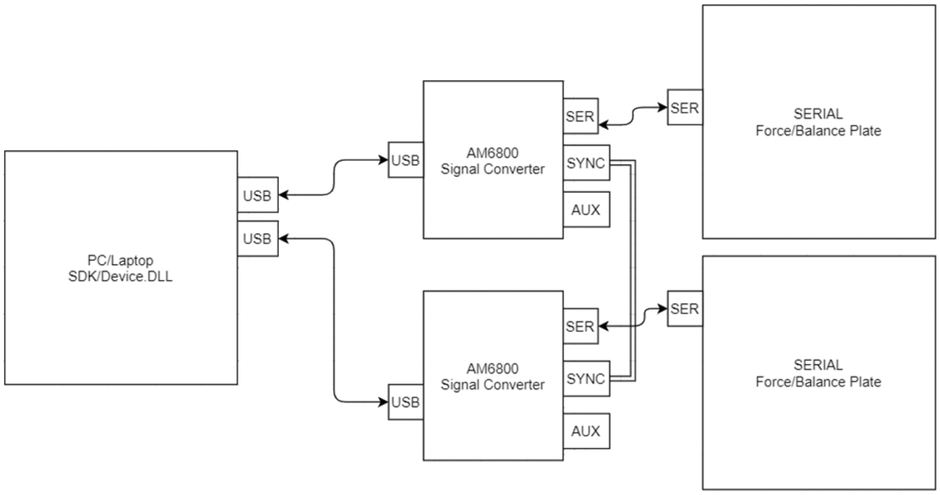
The resampler logic inside the Library will perform the correct downsample/upsample to match the external clock pulses. The clock frequency can be as low as 1Hz and as high as 4000Hz.

The sample file **ExternalClockingFZReaderExample.cpp** shows this in action.

With External clocking, your code cannot directly control the SYNC pin frequency by calling `bertec_SetFrequencyGeneration`. Calling the function while in this mode will result in nothing happening since the SYNC pin is an input, not an output. To control the frequency of the input signal your application will need to either control the source of the clock signals, or provide a method for the user to manually interact with the device.

MULTIPLE AMPLIFERS, SELF-SYNCING

With multiple AM6500/AM6800 amplifiers connected together via the SYNC pin, it becomes possible to synchronize the devices together using the same clock source. In this use case, the clock source is one of the amplifiers, behaving in the same manner as the *Internal Clock Resampling, Single Plate* use case while the other amplifier(s) are set up to behave in the manner as *External Clock Resampling, Single Plate*. The code pattern is exactly the same as *Internal Clock Resampling* with a single device – the Library detects multiple devices and takes care of the rest.



For this use case, you would want to pass `bertec_DataStreamControl::SYNCPINMODE_INTCLOCK` to the `syncPinMode` field, along with a desired generation frequency for the device. For example, this will set the first device to generate a 500Hz signal (again, the code differences from the generic case are highlighted):

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

void myStausCallback(bertec_Handle hand, int status, void * user)
{
  if (status == BERTEC_DEVICES_READY)
  { // start the data stream
    bertec_DataStreamControl streamControl = { 0 };
    streamControl.size = sizeof( streamControl );
    streamControl.syncPinMode = bertec_DataStreamControl::SyncPinMode::SYNCPINMODE_INTCLOCK;
    streamControl.internalClockSource = 0; // set the first device to be the clock source
    streamControl.internalClockFrequency = 500; // 500 Hz
    bertec_StartDataStream( handle, &streamControl );
  }
}

bertec_Handle handle = bertec_Init();
bertec_RegisterDataStreamCallback(handle, myDataCallback, NULL);
bertec_RegisterStatusCallback(handle, myStausCallback, NULL);
bertec_Start(handle);

...your main program runs; the status callback starts the data streaming...

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

This code will set up the device to generate 500Hz pulse signals on the SYNC pin, which the amplifier will read back into the data stream on both amplifiers' `syncData` value. The resampler logic inside the system will use this to 'downsample' the data into an apparent 500Hz data rate. The sample file **InternalClockingFZReaderExample.cpp** shows this in action.

As with the single device *Internal Clock Resampling*, you can change the rate after the fact by using the function `bertec_SetFrequencyGeneration` with the new value. For example:

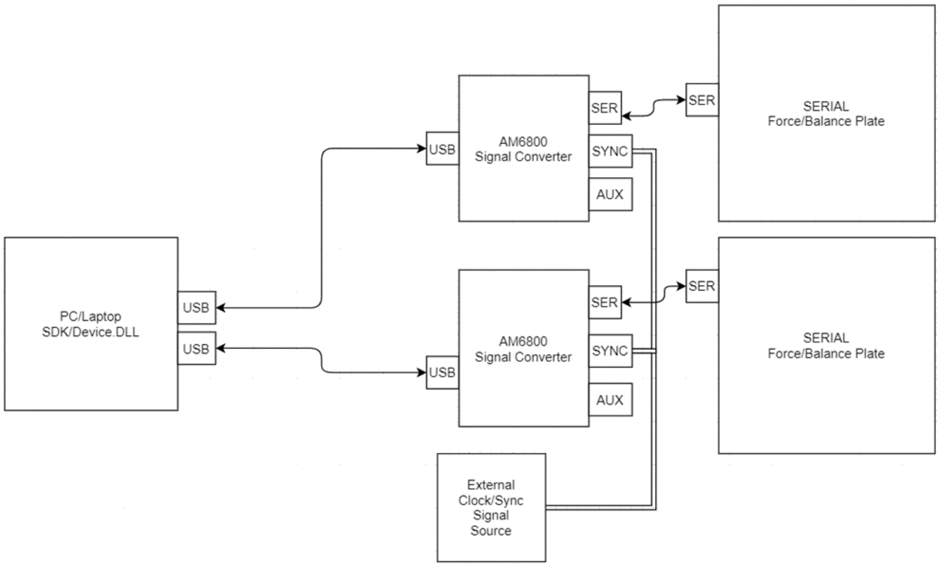
```
bertec_SetFrequencyGeneration( handle, 0, bertec_IOPins::IO_PIN_SYNC, 2000);
```

Will change the SYNC pin output and resulting resampling rate from the above 500Hz to 2000Hz. The data *frame rate* from the hardware is still fixed at 1000Hz, but the data is now being resampled into 2000Hz.

Special note: when using Internal Clock, you *must* set the `internalClockFrequency` value to a valid value (such as 250 or 1000); setting this to zero or a negative number will result in an error.

MULTIPLE AMPLIFERS, EXTERNAL CLOCK

As with *Multiple Amplifiers, Self-Syncing*, connecting multiple AM6500/AM6800 together via the SYNC pin allows the Library to synchronize the devices together using the same clock source. Unlike Self-Syncing, this use case is where the clock source is an external device as in *External Clock Resampling, Single Plate*. The code pattern is exactly the same as *External Clock Resampling*. A typical use for such a setup would be cameras that provide a reference clock pulse.



For this use case, you would want to pass `bertec_DataStreamControl::SYNCPINMODE_EXTCLOCK` to the `syncPinMode` field. For example:

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

void myStausCallback(bertec_Handle hand, int status, void * user)
{
  if (status == BERTEC_DEVICES_READY)
  { // start the data stream
    bertec_DataStreamControl streamControl = { 0 };
    streamControl.size = sizeof( streamControl );
    streamControl.syncPinMode = bertec_DataStreamControl::SyncPinMode::SYNCPINMODE_EXTCLOCK;
    // Note there is no frequency being set
    bertec_StartDataStream( handle,&streamControl);
  }
}

bertec_Handle handle = bertec_Init();
bertec_RegisterDataStreamCallback(handle, myDataCallback, NULL);
bertec_RegisterStatusCallback(handle, myStausCallback, NULL);
bertec_Start(handle);

...your main program runs; the status callback starts the data streaming...

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

As you can see, the code above is exactly the same as *External Clock Resampling, Single Plate*. The data streamer will start to deliver data once the external clock device delivers clock pulses on the SYNC pins *after* there is a ‘quiet’ period where the external clock is held low. If this held-low phase is not implemented, then `StartDataStream` will return an error. The Library uses this held-low period and the start of the clock pulses to determine the data alignment from the devices.

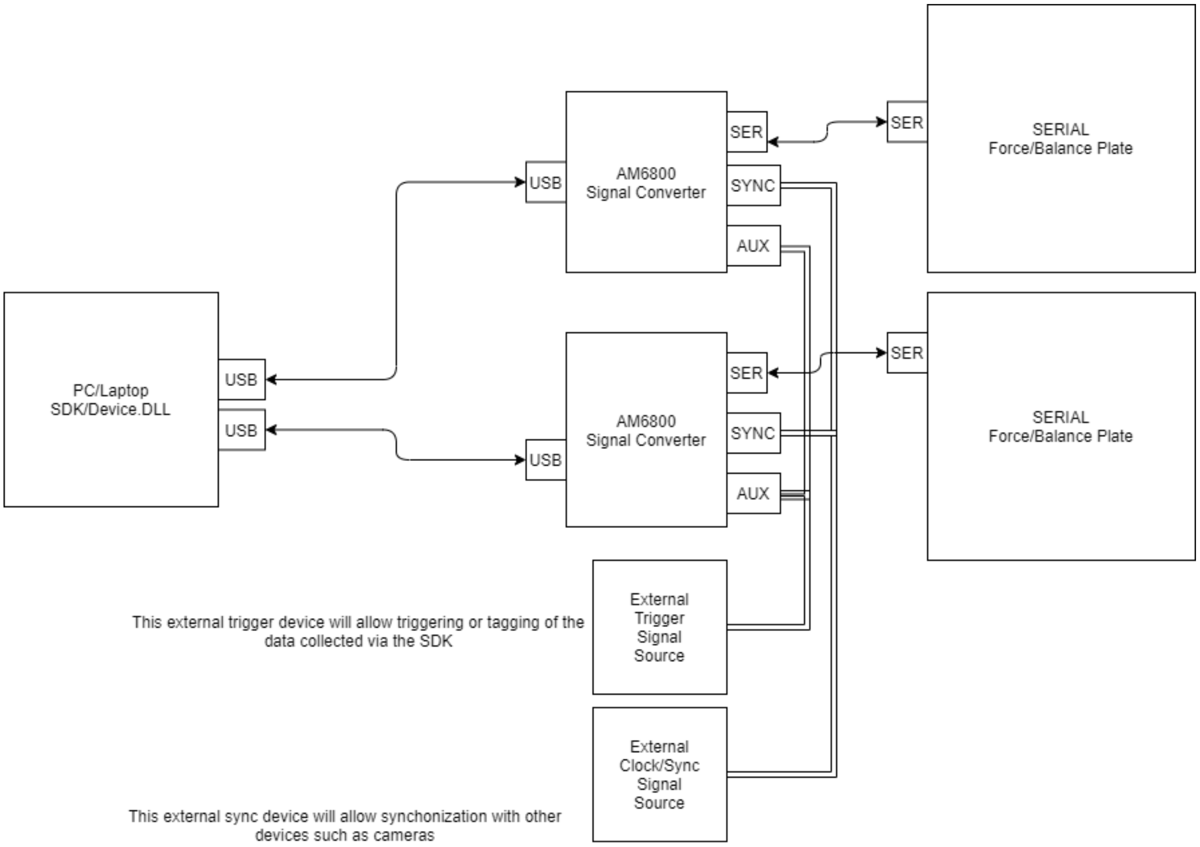
The resampler logic inside the Library will perform the correct downsample/upsample to match the external clock pulses. The clock frequency can be as low as 1Hz and as high as 4000Hz.

The sample file **ExternalClockingFZReaderExample.cpp** shows this in action.

With External clocking, your code cannot directly control the SYNC pin frequency by calling `bertec_SetFrequencyGeneration`. Calling the function while in this mode will result in nothing happening since the SYNC pin is an input, not an output. To control the frequency of the input signal your application will need to either control the source of the clock signals, or provide a method for the user to manually interact with the device.

MULTIPLE AMPLIFERS, EXTERNAL CLOCK, EXTERNAL AUX LINE TRIGGERING

As an alternative to *Multiple Amplifiers, External Clock,* you can elect to connect the SYNC and AUX pins of multiple devices together where the SYNC line connection forms the external clock source, and the AUX line controls the data start/stop signaling. This allows for the eternal clock source to be continually running prior to the AM6500/AM6800 hardware reaching a power-on state and provides for independent data flow control via the AUX line. The setup for this is similar to both *External Clock Resampling* and *Multiple Amplifiers, External Clock*, with the addition of a hardware connection between all the AUX pins and a slight code change to the setup of the Data Stream Control Block. A typical use for such a setup would be a camera system that has a free-running clock source with an external trigger such as a flash detector or hardware push button.



Note the extra AUX line connections in the above image.

For this use case, you need to pass `bertec_DataStreamControl::SYNCPINMODE_EXTAUXCONTROL` to the `syncPinMode` field, and `bertec_DataStreamControl::AUXPINMODE_START_PULSE_ONOFF` to the `auxPinMode` field. For example:

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

void myStausCallback(bertec_Handle hand, int status, void * user)
{
  if (status == BERTEC_DEVICES_READY)
  { // start the data stream
    bertec_DataStreamControl streamControl = { 0 };
    streamControl.size = sizeof( streamControl );
    streamControl.syncPinMode = bertec_DataStreamControl::SyncPinMode::SYNCPINMODE_EXTAUXCONTROL;
    streamControl.auxPinMode = bertec_DataStreamControl::SyncPinMode:: AUXPINMODE_START_PULSE_ONOFF;
    // Note there is no frequency being set
    bertec_StartDataStream( handle,&streamControl);
  }
}

bertec_Handle handle = bertec_Init();
bertec_RegisterDataStreamCallback(handle, myDataCallback, NULL);
bertec_RegisterStatusCallback(handle, myStausCallback, NULL);
bertec_Start(handle);

...your main program runs; the status callback starts the data streaming...

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

As you can see, the code above is exactly the same as *External Clock Resampling, Single Plate* and *Multiple Amplifiers, External Clock,* with the exception of the code highlighted in bold. The data streamer will start to deliver data once the AUX pin has received a high-low transition pulse; prior to this, the AUX pin is expected to be held low. If this held-low phase is not implemented, then `StartDataStream` will return an error. The Library uses this held-low period and the AUX pin high-low transition to determine the data alignment from the devices. Transitions as short as 10ms are allowed.

The resampler logic inside the Library will perform the correct downsample/upsample to match the external clock pulses. The clock frequency can be as low as 1Hz and as high as 4000Hz.

The sample file **InternalClockingFZReaderExampleMultipleSync.cpp** shows this in action.

BERTEC DEVICE LIBRARY FUNCTIONS

All Library functions are exported as “C” 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 application must call this function prior to using any other method beyond `bertec_LibraryVersion` – this includes 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_DATASTREAMCALLBACK TYPEDEF

BERTEC_REGISTERDATASTREAMCALLBACK

BERTEC_UNREGISTERDATASTREAMCALLBACK

`void bertec_DataStreamCallback(bertec_Handle theHandle, const bertec_DataFrame * data, void * userData)`

`int bertec_RegisterDataStreamCallback(bertec_Handle theHandle, bertec_DataStreamCallback fn, void * userData)`

`int bertec_UnregisterDataStreamCallback(bertec_Handle theHandle, bertec_DataStreamCallback fn, void * userData)`

To use the data callback functionality as provided by the Library, you will need to register your callback function with `bertec_RegisterDataStreamCallback`. Only one data 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_UnregisterDataStreamCallback`. Calling `bertec_Close` will automatically unregister all callbacks as will calling `bertec_Init`.

In order to properly unregister the callback, you must call `bertec_UnregisterDataStreamCallback` 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 ensure 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 process the data block as quickly as possible and return.

Refer to the Data Processing and Format section for more information about the format and type of the data block.

BERTEC_IMMEDIATEDEVICEDATACALLBACK TYPEDEF

BERTEC_REGISTERIMMEDIATEDEVICEDATACALLBACK

BERTEC_UNREGISTERIMMEDIATEDEVICEDATACALLBACK

```
void bertec_ImmediateDeviceDataCallback(bertec_Handle theHandle, int deviceIndex, const char*
uid, const bertec_DeviceData * data, void * userData)

int bertec_RegisterImmediateDeviceDataCallback(bertec_Handle theHandle,
bertec_ImmediateDeviceDataCallback fn, void * userData)

int bertec_UnregisterImmediateDeviceDataCallback(bertec_Handle theHandle,
bertec_ImmediateDeviceDataCallback fn, void * userData)
```

This is for special use case functionality and is typically not needed for most code implementations.

Registers callbacks for a single device's data prior to any processing, such as resampling, averaging, filtering, or computed channels. Zeroing offsets (`bertec_ZeroNow, bertec_SetEnableAutozero`) are handled prior to this callback being invoked.

Each time the function registered to the callback is called, the function will get the device index, the unique ID of the device, and a single block of data. The device index is a zero-based value that is the same as the index values passed to other methods such as `bertec_GetDeviceInfo`. The unique ID (`uid`) value is a null-terminated string that can be used to further differentiate the device and is unique to that device and is *not* the same as a serial number. See `bertec_GetDeviceIDString` for more information.

This callback will be called at the data rate of the device (1000hz), and the ordering of the callback between devices is not guaranteed (ex: you may get device # 1, 1, 2, 3, 3, 2, 1, 1 etc.). However, data ordering within the device itself *is* guaranteed (ex: you will get block #1, 2, 3, 4, and never 1, 4, 3, 2 for example). If there is a connection issue, then data blocks may be skipped - this can be detected by a discontinuous `additionalData.timestamp` value. The timestamp value comes from the device itself and increments at a monotonic rate (ex: 1,2,3,4 - a pattern of 1,2,4 indicates block #3 was dropped between the device and the PC).

This callback is designed to be used as a feature where your application needs to have some sort of "monitoring" of the device data stream outside of normal data processing.

Note: this callback is invoked within the context of the low-level USB interface thread - your implementation must return as quickly as possible to avoid data loss.

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 status 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 ensure 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 as 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`. Only one sorting 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_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 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 `bertec_DeviceInfo::serial` value, which is serial number string 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 the contents to reflect your new device order.

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_STARTDATASTREAM

BERTEC_STARTDATASTREAMASYNC

BERTEC_STARTDATASTREAMNOTIFCATION

```
int bertec_StartDataStream( bertec_Handle theHandle, const bertec_DataStreamControl*
pControlStruct )
```

```
int bertec_StartDataStreamAsync( bertec_Handle theHandle, const bertec_DataStreamControl*
pControlStruct, bertec_StartDataStreamNotifcation status_notifcation, void * userData )
```

```
void bertec_StartDataStreamNotifcation(bertec_Handle theHandle, const bertec_DataStreamControl *
control, int status, void * userData)
```

The Library will not deliver data on the DataStream callback or DataStream buffer polling until you call `bertec_StartDataStream` or `bertec_StartDataStreamAsync` with the desired mode (data *will* be delivered to the `bertec_ImmediateDeviceDataCallback` set by `bertec_RegisterImmediateDeviceDataCallback` with or without calling `bertec_StartDataStream`). Calling `bertec_StartDataStream` or `bertec_StartDataStreamAsync` will set up the desired data streaming mode for the connected devices, and limits which devices are read from. The Library does all the needed checks on incoming data and SYNC/AUX pin signals, only returning once the expected conditions are met.

For information on how `StartDataStream` functions and the prerequisite conditions it enforces, see the *Data Streaming and Multiple Devices* section of this document.

Calling `bertec_StartDataStream` or `bertec_StartDataStreamAsync` while a data stream is already started has the same effect of calling `bertec_StopDataStream` first.

Some of the functionality here overlaps with other API methods, such as `bertec_SetExternalClockMode`, but provides a different level of control.

Calling `bertec_StartDataStream` or `bertec_StartDataStreamAsync` with `SYNCPINMODE_NONE` and `AUXPINMODE_NONE` as the control parameters will deliver data without any additional processing or control – this is considered ‘classic’ mode, and will make the Library behave as it did in previous versions.

Note: Outside of `SYNCPINMODE_NONE`, `SYNCPINMODE_CLASSIC`, and `AUXPINMODE_NONE`, sync and aux pin functionally requires updated firmware and matching hardware to function. If outdated firmware or unsupported hardware is used, the Library will reject the requested mode type and return an error. See the `bertec_DeviceInfo` structure, `hasAuxSyncPins` flag.

Note: Starting the data stream is only valid ***after*** devices have been detected; your application’s code should check the status of the connection by either polling `bertec_GetStatus` or using `bertec_StatusCallback` to check for `BERTEC_DEVICES_READY`. Once the devices are ready, then you can start data streaming. See the Example code for how to do this.

The `bertec_StartDataStream` function is blocking non-async method and only returns once the required conditions have been met. `bertec_StartDataStreamAsync` on the other hand will return immediately and the results of the call will be passed through the `status_notifcation` callback pointer.

The `status_notifcation` callback handler will get a pointer to the control struct in use, the `bertec_StatusErrors` status of the data stream, and a user-defined pointer (typically cast to a C++ object or 'this'). While the data stream is being set up, `BERTEC_FUNCTION_BUSY` will be sent multiple times to the callback handler. A successfully setup of the data stream will result in an `BERTEC_NOERROR` result; anything other than `BERTEC_NOERROR` or `BERTEC_FUNCTION_BUSY` should be considered an error.

BERTEC_DATASTREAMCONTROL::SYNCPINMODE ENUM VALUES

State	**Value**	**Explanation**
`SYNCPINMODE_NONE`	0x00	The default state; the devices do no synchronization and do not handle the external clock signal. All devices' SYNC pin modes will be set to `SYNC_IN_SAMPLED`. This is the only mode supported on devices that do not have a SYNC pin.
`SYNCPINMODE_CLASSIC`	0x01	This is a basic mode and will work with older firmware; one device delivers the master reference clock on the SYNC pin and the other devices respond to it. In this mode it is not possible to start or stop the data stream or resample the incoming data.
`SYNCPINMODE_INTCLOCK`	0x02	One device is generating a clock on the SYNC pin, taking the place of the external clock. Data cannot be start or stopped based on the clock signal, but the frequency can be changed via `bertec_SetFrequencyGeneration`. The `bertec_DataStreamControl::internalClockSource` index value must be set to the device that will perform the frequency generation; this device will have it's SYNC pin mode set to `SYNC_OUT_FREQGEN` and all other device's SYNC pin mode set to `SYNC_IN_SAMPLED`. Requires updated firmware; `bertec_StartDataStream` will return an error if this is not supported.
`SYNCPINMODE_EXTCLOCK`	0x03	The devices are getting an external clock connected to the SYNC pins, and the Library will resample the data to match. Data can be started and stopped based on the external clock input. All devices' SYNC pin modes will be set to `SYNC_IN_SAMPLED`. Requires updated firmware; `bertec_StartDataStream` will return an error if this is not supported.
`SYNCPINMODE_INTAUXCONTROL`	0x04	One device is generating a clock on the SYNC pin, taking the place of the external clock. Data cannot be start or stopped based on the clock signal, but the frequency can be changed via `bertec_SetFrequencyGeneration`. The `bertec_DataStreamControl::internalClockSource` index value

		must be set to the device that will perform the frequency generation; this device will have it's SYNC pin mode set to `SYNC_OUT_FREQGEN` and all other device's SYNC pin mode set to `SYNC_IN_SAMPLED`. Requires updated firmware; `bertec_StartDataStream` will return an error if this is not supported. Unlike `SYNCPINMODE_INTCLOCK`, this mode will instead reference the AUX pin for the data control, allowing for the SYNC line to immediately generate pulses while the AUX is externally controlled. The `bertec_DataStreamControl::auxPinMode` field is used to determine when the data is to start/stop and the quiet period high or low state. The `bertec_DataStreamControl::auxPinMode` field must not be set to `AUXPINMODE_NONE` when this mode selected; doing so will result in a error return and the stream will not start.
`SYNCPINMODE_EXTAUXCONTROL`	0x05	The devices are getting an external clock connected to the SYNC pins, and the Library will resample the data to match. Data can be started and stopped based on the external clock input. All devices' SYNC pin modes will be set to `SYNC_IN_SAMPLED`. Requires updated firmware; `bertec_StartDataStream` will return an error if this is not supported. Unlike `SYNCPINMODE_EXTCLOCK`, this mode will instead reference the AUX pin for the initial 'quiet period' which allows for the SYNC line to be continually fed without holding it low (for example, an external clock source that cannot be paused). The `bertec_DataStreamControl::auxPinMode` field is used to determine when the data is to start/stop and the quiet period high or low state. The `bertec_DataStreamControl::auxPinMode` field must not be set to `AUXPINMODE_NONE` when this mode selected; doing so will result in a error return and the stream will not start.
`SYNCPINMODE_RUNHIGH`	0x20	The data stream will only be delivered when the SYNC pin is at a high (1) state. All devices' SYNC pin modes will be set to `SYNC_IN_SAMPLED`. Requires updated firmware; `bertec_StartDataStream` will return an error if this is not supported.
`SYNCPINMODE_RUNLOW`	0x21	The data stream will only be delivered when the SYNC pin is at a low (0) state. All devices' SYNC pin modes will be set to `SYNC_IN_SAMPLED`. Requires updated firmware; `bertec_StartDataStream` will return an error if this is not supported.

`SYNCPINMODE_START_PULSE_ON`	0x22	The data stream will only start once there is a low-high-low transition on the SYNC pin. Once triggered, the data stream will not stop and will continue to run; additional pulses will be ignored. All devices' SYNC pin modes will be set to `SYNC_IN_SAMPLED`. Requires updated firmware; `bertec_StartDataStream` will return an error if this is not supported.
`SYNCPINMODE_START_PULSE_ONOFF`	0x23	The data stream will only start once there is a low-high-low transition on the SYNC pin, and then continue to run until the next low-high-low transition. Each pulse pair will cause data to start and then stop. All devices' SYNC pin modes will be set to `SYNC_IN_SAMPLED`. Requires updated firmware; `bertec_StartDataStream` will return an error if this is not supported.

BERTEC_DATASTREAMCONTROL::AUXPINMODE ENUM VALUES

State	**Value**	**Explanation**
`AUXPINMODE_NONE`	0x00	The default state; the aux pin is not treated special in any way. This is the only mode supported with `SYNCPINMODE_CLASSIC`. All devices' AUX pin modes will be set to `AUX_NONE_AUX_IN_ZERO` or `AUX_IN_SAMPLED` depending on firmware. This is the only mode supported on devices that do not have an AUX pin.
`AUXPINMODE_RUNHIGH`	0x20	The data stream will only be delivered when the AUX pin is at a high (1) state. All devices' AUX pin modes will be set to `AUX_IN_SAMPLED`. Requires updated firmware; `bertec_StartDataStream` will return an error if this is not supported.
`AUXPINMODE_RUNLOW`	0x21	The data stream will only be delivered when the SYNC pin is at a low (0) state. All devices' AUX pin modes will be set to `AUX_IN_SAMPLED`. Requires updated firmware; `bertec_StartDataStream` will return an error if this is not supported.
`AUXPINMODE_START_PULSE_ON`	0x22	The data stream will only start once there is a low-high-low transition on the SYNC pin. Once triggered, the data stream will not stop and will continue to run; additional pulses will be ignored. All devices' AUX pin modes will be set to `AUX_IN_SAMPLED`.

		Requires updated firmware; `bertec_StartDataStream` will return an error if this is not supported.
`AUXPINMODE_START_PULSE_ONOFF`	0x23	The data stream will only start once there is a low-high-low transition on the AUX pin, and then continue to run until the next low-high-low transition. Each pulse pair will cause data to start and then stop. All devices’ AUX pin modes will be set to `AUX_IN_SAMPLED`. Requires updated firmware; `bertec_StartDataStream` will return an error if this is not supported.

BERTEC_DATASTREAMCONTROL STRUCTURE

Field	**Type**	**Explanation**
`size`	int	Size of the control structure; must be set to sizeof(`bertec_DataStreamControl`). Setting this to zero will result in the data stream using `SYNCPINMODE_NONE` and `AUXPINMODE_NONE`.
`syncPinMode`	SyncPinMode	Controls how the hardware SYNC pin is used. See the `bertec_DataStreamControl::SyncPinMode` enum.
`auxPinMode`	AuxPinMode	Controls how the hardware AUX pin is used. See the `bertec_DataStreamControl::AuxPinMode` enum.
`internalClockSource`	int	Device index of the internal clock source, if any. Only used when `syncPinMode` = `SYNCPINMODE_INTCLOCK`.
`internalClockFrequency`	float	The frequency to be generated by the internal clock source. Only used when `syncPinMode` = `SYNCPINMODE_INTCLOCK` and `internalClockSource` is a valid index (0 to total # of devices – 1)
`deviceFilterBitmask`	unsigned int	Used to control which devices are delivering data via the `bertec_DataStreamCallback` or `bertec_ReadBufferedDataStream`. Each bit corresponds to a device index; bit 0 == device index 1, bit 1 == index 1, etc. Setting this mask to a zero (all bits off) is treated the same as all bits on (0xFFFFFFFF). It is entirely possible to set the `internalClockSource` value to a device that has been masked out by `deviceFilterBitmask` – doing

		effectively turns the masked device into an external clock source. Setting `deviceFilterBitmask` to allow only 1 device through is not an error. NOTE: your application will need to handle mapping the resulting data frame to any real device index.

BERTEC_STOPDATASTREAM

`int bertec_StopDataStream( bertec_Handle theHandle)`

Stops the current data stream and returns the hardware back to their default states. This will have the effect of resetting both the syncPinMode and auxPinMode modes to NONE, clearing the device filter bitmask, and setting the external clock mode back to internal (no resampling) for all devices.

The Data Stream callback and polling buffers will no longer receive any data until either `bertec_StartDataStream` or `bertec_StartDataStreamAsync` are called.

BERTEC_GETCURRENTDATASTREAMCONTROL

`int bertec_GetCurrentDataStreamControl( bertec_Handle theHandle, bertec_DataStreamControl* pControlStructOut, size_t structOutSize )`

Fills the passed `bertec_DataStreamControl` structure with a copy of the current control data set by `bertec_StartDataStream` or `bertec_StartDataStreamAsync`; if there is no data stream currently set up this will return an empty copy.

The `structOutSize` parameter must be equal to the size of the `bertec_DataStreamControl` structure; if this is not a match, the function call will return an 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_ReadBufferedDataStream` 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_READBUFFEREDDATASTREAM

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

Instead of using the callbacks, you can use the `bertec_ReadBufferedDataStream` 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_ReadBufferedDataStream` will take one data frame from the internal buffer, copying the values into your passed pointer and returning a success or failure result code. A return value of 1 indicates the data was copied from the internal buffer to your passed `dataFrame` memory block – there may or may not be more data available (see `bertec_GetBufferedDataAvailable`, above). A return value of 0 indicates that there was no data left in the buffer and your passed `dataFrame` memory block was not changed. Any other return value is considered and error and should be handled accordingly.

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 );
```

`DeviceCount` can either be the current number of connected devices or some arbitrarily large number. As long as the buffer space is large enough, `bertec_ReadBufferedDataStream` will fill it with the buffered data.

As a convenience, the Library provides `bertec_AllocateReadBufferedData` and `bertec_FreeAllocatedReadBufferedData` which your application can use to allocate an appropriate sized buffer for the currently connected devices. The buffer allocation is processor-aligned, typically on 8-byte boundaries (needed for some CPU architectures).

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

Negative results from `bertec_ReadBufferedDataStream` 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_ALLOCATEREADBUFFEREDDATA

BERTEC_ALLOCATEREADBUFFEREDDATAFORCOUNT

BERTEC_FREEALLOCATEDREADBUFFEREDDATA

```
bertec_DataFrame * bertec_AllocateReadBufferedData( bertec_Handle bHand, size_t*
dataFrameSizeOut )
bertec_DataFrame * bertec_AllocateReadBufferedDataForCount( bertec_Handle bHand, int
deviceCount, size_t* dataFrameSizeOut );
int bertec_FreeAllocatedReadBufferedData( bertec_Handle bHand, bertec_DataFrame* dataFrame )
```

To facilitate create and using `bertec_ReadBufferedDataStream`, three convenience functions have been provided. Calling `bertec_AllocateReadBufferedData` will allocate a buffer large enough to handle the currently connected number of devices – if the device count changes, you will need to release the memory and re-allocate it prior to calling `bertec_ReadBufferedDataStream`.

Optionally, you may call `bertec_AllocateReadBufferedDataForCount`, passing a `deviceCount` value to pre-allocate for a fixed number of devices instead of using the current device count. This method is preferred when the device count is expected to change, or you wish to pre-allocate data prior to any connection event.

The allocated memory is processor-aligned, typically on 8-byte boundaries (needed for some CPU architectures).

The allocated memory must be released when you are done with it, typically when unloading the Library or at application termination; failing to do so will result in memory leaks.

Call `bertec_FreeAllocatedReadBufferedData` with the allocated memory pointer to release the memory block.

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. There is a hard limit of 10,000 for the `newMaxSamples` value, which is equivalent to 10 seconds worth of data.

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 `BERTEC_NOERROR` if successful.

`deviceIndex` should be from 0 to one less than the number of connected devices; passing a value outside this range will result in a return code of `BERTEC_INDEX_OUT_OF_RANGE` and leave the `info` pointer contents untouched.

If the `info` pointer is null or the `infoSize` value is less than the size of the `bertec_DeviceInfo` structure, then this function will return `BERTEC_INVALID_PARAMETER`.

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. Returns `BERTEC_NOERROR` if successful.

`deviceIndex` should be from 0 to one less than the number of connected devices; passing a value outside this range will result in a return code of `BERTEC_INDEX_OUT_OF_RANGE` and leave the `buffer` pointer contents untouched.

If the `buffer` pointer is null or the `bufferSize` value is less than 1, then this function will return `BERTEC_INVALID_PARAMETER`.

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 serial number directly instead of reading the entire device info into a `bertec_DeviceInfo` struct and accessing the `model` member. Returns `BERTEC_NOERROR` if successful.

`deviceIndex` should be from 0 to one less than the number of connected devices; passing a value outside this range will result in a return code of `BERTEC_INDEX_OUT_OF_RANGE` and leave the `buffer` pointer contents untouched.

If the `buffer` pointer is null or the `bufferSize` value is less than 1, then this function will return `BERTEC_INVALID_PARAMETER`.

BERTEC_GETDEVICEIDSTRING

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

This function will return the hardware's unique device id string, which is typically an 8-character alphanumeric string. Possible examples are A2QYVYSD and NS94ASM8. This is the same value that is passed into `bertec_ImmediateDeviceDataCallback` and `bertec_DeviceTimestampCallback`. Returns `BERTEC_NOERROR` if successful.

`deviceIndex` should be from 0 to one less than the number of connected devices; passing a value outside this range will result in a return code of `BERTEC_INDEX_OUT_OF_RANGE` and leave the `buffer` pointer contents untouched.

If the `buffer` pointer is null or the `bufferSize` value is less than 1, then this function will return `BERTEC_INVALID_PARAMETER`.

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` structure and accessing the `channelNames` member. The channel names are copied into the buffer parameter 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`".

This function will return the number of channels from the `bertec_DeviceInfo` structure and accessing the `channelCount` member.

`deviceIndex` should be from 0 to one less than the number of connected devices; passing a value outside this range will result in a return code of `BERTEC_INDEX_OUT_OF_RANGE` and leave the `buffer` pointer contents untouched.

If the `buffer` pointer is null or the `bufferSize` value is less than 1, then this function will return `BERTEC_INVALID_PARAMETER`.

BERTEC_GETDEVICECHANNELCOUNT

`int bertec_GetDeviceChannelCount( bertec_Handle bHand, int deviceIndex )`

This is a convenience function that will return the number of channels for the given `deviceIndex` number. You can use this function instead of reading the entire device info into a `bertec_DeviceInfo` struct and accessing the `channelCount` member. This channel count (like the `bertec_DeviceInfo::channelCount` value) includes the optional computed channels (Center of Pressure, Center of Gravity, and Sway) if they have been enabled (see `bertec_SetComputedChannelsFlags`).

`deviceIndex` should be from 0 to one less than the number of connected devices; passing a value outside this range will result in a return code of `BERTEC_INDEX_OUT_OF_RANGE`.

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. 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). Available channels will include the device’s hardware channels and any computed channels, if enabled (see `bertec_SetComputedChannelsFlags`).

`deviceIndex` should be from 0 to one less than the number of connected devices, and the `channelIndex` must be from 0 to one less than that device’s channel count; passing values outside of either of these ranges will result in a return code of `BERTEC_INDEX_OUT_OF_RANGE` and leave the `buffer` pointer contents untouched.

If the `buffer` pointer is null or the `bufferSize` value is less than 1, then this function will return `BERTEC_INVALID_PARAMETER`.

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`:

BERTEC_AUTOZEROSTATES ENUM VALUES

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_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 VALUES

Enum Name	**Value**	**Description**

`SYNC_IN_SAMPLED`	0x00	The SYNC pin is an input, but its value is not interpreted in any way. This mode is also known as `SYNC_NONE`. This is the default power-up mode. The SYNC pin is sampled at a 8kHZ rate.
`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_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 `bertec_SetSyncAuxPinValues` function.
`SYNC_OUT_FREQGEN`	0x09	The SYNC pin is acting as a frequency generator. The frequency is set via the `bertec_SetFrequencyGeneration` function.

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 `bertec_AuxModeFlags` value, 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 ENUM VALUES

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. The AUX pin is sampled at a 8kHZ rate.
`AUX_OUT_INSTANT`	0x02	The AUX is outputting the value most recently set via the `bertec_SetSyncAuxPinValues` function.
`AUX_OUT_PATGEN`	0x04	The AUX pin is outputting a random pattern. This is useful for debugging.

BERTEC_SETPINMODE

`int bertec_SetPinMode( bertec_Handle bHand, int deviceIndex, bertec_IOPins pin, bertec_PinModes newMode)`

Sets the given `bertec_IOPins` enum to the selected `bertec_PinModes` mode. This is only valid for devices that support the extended SYNC and AUX feature set. Only one pin at a time can be set - if you desire to set multiple pins to the same or different modes, you must call this function multiple times with different selected pin enum values.

Calling this with the pin parameter set to set to `bertec_IOPins.IO_PIN_NONE` will result in an error.

BERTEC_IOPINS ENUM VALUES

Enum Name	**Value**	**Description**
`IO_PIN_SYNC`	0x00	The SYNC pin, typically used for synchronizing data samples. Available on AM6500, AM6800, AM6817 amplifiers.
`IO_PIN_AUX`	0x01	The AUX/ZERO pin, used for general-purpose I/O. Available on AM6500 amplifiers as the bidirectional AUX pin, and on AM6800 and AM6817 amplifiers as the input-only ZERO pin
`IO_PIN_CH7`	0x06	The CH7 output pin, available on AM6817E and higher amplifiers. Not available on AM6500 amplifiers.
`IO_PIN_CH8`	0x07	The CH8 output pin, available on AM6817E and higher amplifiers. Not available on AM6500 amplifiers.
`IO_PIN_NONE`	0xFF	Special value only used in query requests; results in an error if used to set a pin.

BERTEC_PINMODES ENUM VALUES

Enum Name	**Value**	**Description**
`PINMODE_IN_SAMPLED`	0x00	The pin is an input, and its value is not interpreted in any way.
`PINMODE_OUT_SYNC_MARK`	0x01	The pin is outputting a 1kHz square wave clock with a reference mark embedded every 2000ms.
`PINMODE_IN_SYNC_MARK`	0x02	The pin is inputting a 1kHz square wave clock with optional reference marks.
`PINMODE_OUT_RANDPAT`	0x04	The pin is outputting a random pattern. This is useful for debugging.
`PINMODE_IN_CONTINUOUS`	0x05	The pin is inputting a continuous 1kHz square wave clock without reference marks.
`PINMODE_OUT_CONTINUOUS`	0x07	The pin is outputting a continuous 1kHz square wave clock without reference marks.
`PINMODE_OUT_INSTANT`	0x08	The pin is outputting the value most recently set via the OUTPUT command.
`PINMODE_OUT_FREQGEN`	0x09	The pin is acting as a frequency generator, its frequency set via the `bertec_SetFrequencyGeneration` function.
`PINMODE_IN_ZERO`	0x0A	The pin input controls the zeroing of analog outputs. A low logic level keeps the analog output signals zeroed. Only available on available on AM6817E and higher amplifiers.
`PINMODE_OUT_LOAD`	0x0B	The pin outputs the analog value of the load from the transducer. Depends on hardware support for this to work.

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. Only the pin(s) set to `_INSTANT` will be changed; pins not set to `_INSTANT` will ignore this request.

The `syncValue` and `auxValue` parameters are treated as a 0/1 of/on value; passing a value of 0 will turn the pin *off*, and any non-zero value (ex: 1, 16, 255) will turn the pin *on*.

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_RESETDEVICETIMESTAMP

`int bertec_ResetDeviceTimestamp(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_RESETALLDEVICETIMESTAMPS

`int bertec_ResetAllDeviceTimestamps(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_RESETDEVICETIMESTAMPATMARK

`int bertec_ResetDeviceTimestampAtMark(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_RESETALLDEVICETIMESTAMPSATMARK

`int bertec_ResetAllDeviceTimestampsAtMark(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. Depending 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.

BERTEC_CLOCKSOURCEFLAGS ENUM VALUES

Flag Name	**Value**	**Explanation**
`CLOCK_SOURCE_INTERNAL`	0x00	This is the default state and the Library 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, bertec_AggregateDeviceMode newMode)`

`bertec_AggregateDeviceMode 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_AGGREGATEDEVICEMODE ENUM VALUES

Mode Name	**Value**	**Explanation**
`NO_AGGREGATEDEVICEMODE`	0x00	the default mode - no special processing is done
`FRONT_TO_BACK_AGGMODE`	0x01	the plates are arranged length-wise, front to back, with the front plate rotated 180 degrees.

`SIDE_BY_SIDE_AGGMODE`	0x02	the plates are arranged side-by-side (i.e.: treadmill), with the right plate rotated 180 degrees

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`)

BERTEC_COMPUTEDCHANNELFLAGS ENUM VALUES

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_GETSUBJECTHEIGHT

`int bertec_GetSubjectHeight(bertec_Handle bHand, float* heightMMOut)`

Returns the last value set by bet `bertec_SetSubjectHeight` into the `heigthMMOut` pointer. If `bertec_SetSubjectHeight` was never called, then this function will set the float pointed to by `heightMMOut` to zero.

Returns an error if the `heightMMOut` pointer value is null.

BERTEC_DEVICEDATARATE

`float bertec_DeviceDataRate(bertec_Handle bHand, int deviceIndex)`

This function returns the dynamically computed data rate value of the connected devices, in hertz. This value is updated every 2 seconds (2000 ms) per device. On most hardware configurations, this value will typically be around 1000hz; if external clocking mode is enabled for the device (see bertec_SetExternalClockMode), then this value is based on the input signal to the SYNC pin.

This function is provided primarily as a way for any user interface display to show the current data rate from the device. Please note, that due to the way the PC hardware operates, this value will “flutter” approximately ± 5.0hz. Even when using an external clock signal, the devices will always deliver data at a fixed 1000hz.

BERTEC_REDETECTCONNECTEDDEVICES

`int bertec_RedetectConnectedDevices(bertec_Handle bHand)`

Signals that the Library 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. This function is provided primarily to force a redetection of multiple plates that have been added after the SDK has been started. Calling `bertec_RedetectConnectedDevices` does not block and returns immediately.

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.

BERTEC_SETDATARATERESAMPLING

`int bertec_SetDataRateResampling(bertec_Handle bHand, int deviceIndex, int newFrequency)`

Sets and enables the data rate resampling. This will disable both the SYNC and AUX pin modes, returning the device to SAMPLED mode - the SYNC and AUX data values in the Data Frame will always be zero. This feature will work with any device, even if there is no hardware support for SYNC or AUX.

This is an advanced function and is typically not needed for most projects. For systems equipped with AM6500 or AM6800, the preferred method is to use the External Clock Mode and use an externally generated pulse signal to control the data rate resampling. However, for setups without the appropriate hardware this functionality is provided in software.

Data rate resampling is usually done to match the output rate from the device(s) to a separate piece of hardware – for example, a camera running at 125hz.

Passing 0 for `newFrequency` will turn off the data rate resampling and return to the hardware data rate of 1000hz. This will also allow both the SYNC and AUX modes to be changed.

Note that not all frequencies are available; the resulting value is equal to `floor(8000/round(8000/newFrequency))`. Data rate resampling can also introduce delays in the data.

Setting or changing the resampling value will cause momentary fluctuations in the data until the resampling algorithm has collected enough data. Typically, you should discard up to 500 samples after setting or changing this value.

BERTEC_SETFREQUENCYGENERATION

`int bertec_SetFrequencyGeneration( bertec_Handle bHand, int deviceIndex, bertec_IOPins pin, float frequency)`

Sets the frequency generator parameters for a given bertec_`IOPins` enum. The frequency generator is only active if the `bertec_IOPins` pin has been set to `bertec_PinModes.PINMODE_OUT_FREQGEN`. If the hardware does not support frequency generation or the selected pin is not in this mode, the function will return a `BERTEC_UNSUPPORED_COMMAND` error.

Currently, only the SYNC pin supports frequency generation, and the typical range for this is approximately 183hz to 4000hz.

BERTEC_GETFREQUENCYGENERATIONLIMITS

`int bertec_GetFrequencyGenerationLimits( bertec_Handle bHand, int deviceIndex, bertec_IOPins pin, float* frequencyMin, float* frequencyMax )`

Gets the frequency generator min and max limits for a given bertec_`IOPins` enum. If the device or the selected pin does not support frequency generation, then this function will return a `BERTEC_UNSUPPORED_COMMAND` error.

If either the `frequencyMin` or `frequencyMin` parameters are null, then this function will return a `BERTEC_INVALID_PARAMETER` error.

You can use this function to determine if the device and pin supports frequency generation, and a valid range for your application to choose from.

BERTEC_SETUNIFIEDDATAMODE

BERTEC_GETUNIFIEDDATAMODE

`int bertec_SetUnifiedDataMode(bertec_Handle bHand, int enabled)`
`int bertec_GetUnifiedDataMode(bertec_Handle bHand)`

By default, the library will only present data via the callback or data polling when ***all*** devices have data, allowing for software sync, device aggregation, and data averaging. This is typically the desired mode, but some applications may benefit from turning this functionality off.

If the `enabled` parameter is set to 0 (false), then the library will present data whenever ***any*** device has data, even if the other devices do not. Software sync, device aggregation, and data averaging will ***not*** be performed in this mode. The data frame received by the callback or data polling will be incomplete; your implementation must be ready to check for and handle cases where device #1 presents data but #2 will not, and then some frames later that will change to where #2 has data but #1 does not; the data will appear to be unaligned with zero values for the no-data-present device structures.

The simplest method to handle this situation is to check the `bertec_ChannelData::count` value for the `bertec_DeviceData` structure; if this is zero, there is no data for that device in the current frame and that device data can be ignored or dropped. For example:

```
public void onData(bertec_Handle bHand, const bertec_DataFrame * dataFrame, void * userData)
{
  for (int deviceNum = 0; deviceNum < dataFrame->deviceCount; ++ deviceNum)
  {
    const bertec_DeviceData& deviceData = dataFrame->device[devNum];
    int channelCount = deviceData.channelData.count;
    if (channelCount > 0)
        SingleDeviceDataDataHandler(deviceNum, deviceData);
  }
}
```

This example code will only call `SingleDeviceDataDataHandler` when the data frame for the device has channel data. If `UnifiedDataMode` is left on, then this code will still work with no changes required.

Turning off unified data implies that your application will do its own post-processing of the data, using either the sequence number or timestamp values to perform specialized alignment.

Call this function prior to calling Start to ensure the data being received is in the format expected.

Using non-unified data mode with a single plate has no net effect.

Computed channels, device clocks and sync pin settings (`bertec_SetComputedChannelsFlags`, `bertec_ResetDeviceTimestamp`, `bertec_SetExternalClockMode`, etc) will still function if unified data is turned off; however, data averaging and aggregation (`bertec_SetAveraging`, `bertec_SetAggregateDeviceMode`) along with timestamp alignment (`bertec_SetTimestampAlignmentMode`) will not.

BERTEC_SETTIMESTAMPALIGNMENTMODE

BERTEC_GETTIMESTAMPALIGNMENTMODE

```
int bertec_SetTimestampAlignmentMode (bertec_Handle bHand, int enabled)
int bertec_GetTimestampAlignmentMode (bertec_Handle bHand)
```

By default, the library will not attempt to align the incoming device data using the timestamp fields. Calling `bertec_SetTimestampAlignmentMode` with a `TRUE` or `1` value will turn on this feature, telling the library to align the outgoing combined data frame by comparing the incoming timestamps.

Aligning the timestamps may introduce data delays or dropped data frames from one or more devices. It is almost always better to instead use an eternal hardware data marker on the SYNC or AUX channel and look for that in the data frame stream.

Turning *off* the Unified Data Mode will also effectively turn this off, since the data frame is no longer combined.

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 (BERTEC_STATUSERRORS)

Error	Value	Explanation
`BERTEC_NOERROR`	0	Generic no error result.
`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_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_UNABLE_TO_START_STARTED`	-30	`bertec_Start` was called twice; the second call was ignored.
`BERTEC_UNABLE_TO_START_STOPPING`	-31	`bertec_Start` was called while the last `bertec_Stop` call was still being processed; the `bertec_Start` call was ignored.
`BERTEC_UNABLE_TO_STOP_NOTRUNNING`	-32	The library is already stopped or was never running (safe to ignore).
`BERTEC_UNABLE_TO_STOP_STOPPING`	-33	`bertec_Stop` was called twice; the second call was ignored.
`BERTEC_UNABLE_TO_STOP_STARTING`	-34	`bertec_Stop` was called while the last `bertec_Start` call was still being processed; the `bertec_Stop` call may not take effect or take effect later than expected.

`BERTEC_UNABLE_TO_START_FAILED`	-35	Internal error - device threads have failed to start and the Library cannot be used. This is a fatal error.
`BERTEC_UNABLE_TO_STOP_FAILED`	-36	Internal error - device threads have failed to shut down and you may need to hard-kill the application (very unexpected).
`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 devices connected and data is being delivered.
`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_FUNCTION_BUSY`	-203	Either the method or a sub-method is busy - possibly calling the same function from multiple threads.
`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, 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. Version numbers of both C and .NET libraries now synchronized.	Todd Wilson
9/10/2019	2.21	Added support for disabling the unified/combined data mode for multiple plates.	Todd Wilson
2/25/2020	2.35	Added support for polling allocation and clarified various function results and input parameters.	Todd Wilson
4/15/2020	2.40	Updated documentation with new functions.	Todd Wilson
1/22/2021	2.43	Update documentation to cover the pin modes and the frequency generator.	Todd Wilson
3/23/2021	2.44	Added single device data callback functionality.	Todd Wilson
4/14/2021	2.45	Removed the synchronize- and sequence-related enums from bertec_StatusErrors since they are no longer emitted.	Todd Wilson
8/3/2022	2.50	Renamed the Data callback functions to DataStream, and removed the Timestamp manipulating callbacks. Aggregate Device Mode now takes an expanded enum value. Added documentation for the Data Stream Control along with flow diagrams. Removed the 32 device count limit.	Todd Wilson Mohan Baro