# *BarrettHand™*

## BH8 C-Function Library Manual
## Version 4.3x

**Barrett Technology Inc.**

# Table of Contents

# List of Tables

# 1 Introduction

This document describes the BarrettHand Version 4.3 C-Function Library. This Library provides the C++-fluent user with a convenient program interface to the BarrettHand. It is compatible with Firmware Versions 4.3 and newer and is compiled for Windows 95/98/NT/2000 with Microsoft C++. The C-Function Library Version 4.3 includes new RealTime commands as well as some new Supervisory commands introduced as the Firmware was upgraded from 4.1 and 4.2. It is a typical C++ library, providing a `class BHand,` from which you may instantiate multiple Hand objects, each communicating via a unique user-specified COM ports. Before version 4.3, only one Hand could be controlled via the C-Function Library. See Section 4.1.2 for an example of instantiating two hand objects.

Version 4.3 is designed to be compatible with code written for C-Function Library Version 1.0. The Library performs an implicit translation of the old commands into the new format, so you can recompile and link existing DOS programs with the edits listed in Section 3.7. Table 1 summarizes C-Function Library and Firmware compatibility.

**Table 1 – C-Function Library and Firmware Compatibility**

| *C-Function Library Version* | *Firmware Version* |
|---|---|
| 1.0 | 3.0 |
| 4.3x | 4.3x |

The C-Function Library Version 4.3 uses a multithreaded mechanism for accessing serial ports, allowing both synchronous and asynchronous control of the low-level thread and ensuring that all serial communications are executed with high priority. The multithreading mechanism replaces the interrupt-driven processing from C-Function Library Version 1.0. A few of the functions and parameters, including those for multithreading, are advanced and segregated in this document. While they are documented for completeness, we recommend avoiding use of these commands or changing the parameter defaults. Describing the basics of multithreaded processing is beyond the scope of this document – we refer the reader to the extensive documentation provided with the Windows API on the Web.

The following documentation contains three sections:

- Section 2 describes in detail the functional organization of the software, the most important aspects of its operation, and the functions that provide the basic services used by the rest of the C-Function Library.
- Section 3 contains a list of the C-Function Library functions and documentation of their syntax, purpose, inputs, and outputs.
- Section 4 contains example programs that you may find helpful.

# 2 Functional Organization

The C-Function Library is organized into three layers of processing: low, intermediate, and high-level:

1. The low-level processing is responsible for actual communication between the BarrettHand and the host computer. This level runs in a separate thread with the highest priority. It is idle in the absence of communication requests (without consuming CPU time), and is only activated when characters need to be sent and/or received.
2. The intermediate-level processing provides a uniform interface to the programmer for sending and receiving characters. Rather than executing the actual serial data exchange, this level places communication requests to the low-level, and optionally waits for their completion. This level is used by all high-level functions to communicate with the hardware.
3. The high-level processing contains the functions that the user program normally calls (unless you wish to access the lower-level mechanisms directly). These functions allow you to control the hand without having to manage the serial communication.

The design goal behind this layered organization is the following: the low-level processing allows the serial data exchange to have priority over all other tasks, such as refreshing windows, disk read/write, or other user program processing. Delays are minimized and no characters are lost between the host computer and the BarrettHand. At the same time we do not want the user program to run with high priority, because it will execute many other operations unrelated to serial communication, running all of them with high priority. This would significantly increase the response time of the operating system.

The intermediate-level processing provides both synchronous and asynchronous communication modes for user programs, allowing you to send commands to the BarrettHand and have the option to wait for a response or continue with the program. This is not possible with standard Windows serial port function calls, since they require you to decide which mode will be used at the time of initializing the serial port. The high-level processing uses the functionality of both the low-level and intermediate-level processing. This allows the host computer to communicate with the BarrettHand successfully with minimal delays in the response time of the system.

The remaining part of this section describes the three levels in bottom-up order. It closely follows the structure of the `class Bhand` declaration, found in the header file *Bhand.h*, see Appendix A.

## 2.1 Low-Level Processing

Low-level processing is the communication between the BarrettHand and the host computer. This level is transparent, but is included so that you can better understand the Library. This process runs in a separate thread and is given the highest priority. The following function controls the serial communication:

```
DWORD _stdcall _ComThreadFunction( void* pHand )
```

It is started during the initialization phase and contains a single loop, which detects communication requests and executes them. The synchronization between the low-level and the user program thread (containing the two higher levels) is accomplished using the Windows event objects `requestPending` and `requestComplete`. Briefly, the thread function is blocked until the user program signals `requestPending`, at which point the thread function writes data to the serial port and waits for a response. The expected response is determined by the request type (one of `BHREQ_REALTIME, BHREQ_SUPERVISE, BHREQ_EXIT,` or `BHREQ_CLEAR`). When request processing is finished, the thread function signals `requestComplete` and blocks its execution until the next request is detected.

Immediately before the end of processing the request, the thread function calls an optional user-supplied callback function, which can perform a desired periodic activity. Keep in mind that the callback function is executed with

high priority and it should not be computationally intensive. To specify a callback function, set the `pCallback` pointer to its address.

Windows maintains a set of serial timeout parameters (in milliseconds) that determine how long serial port functions should wait before returning with a timeout error. These parameters are set to default values during the initialization of the Library, but you can modify them by using the (intermediate-level) function `ComSetTimeouts`.

Note that the timeout values affect the behavior of the low-level processing thread only. The high-level user functions provide a separate timeout mechanism specifying how long the user program (in synchronous mode) should wait for the low-level thread before it resumes execution.

## 2.2 Intermediate-Level Processing

The functions and variables relevant here are listed in Section 3.4 and the Serial Communications section of the header file, *BHand.h*. Together with the low-level processing thread described in Section 2.1, they represent a self-contained module that can be used to communicate with the hand without ever calling the high-level functions. Even if you intend to use the high-level functions described in Sections 3.2 and 3.3, understanding the material in this subsection is necessary to use some of the more advanced mechanisms provided by the Library.

Note that many functions in the Library return an integer status code. Usually, if the BarrettHand command returns a value and a status code, the Library function will require a pointer to the return value passed as a parameter. Commands that do not pass status codes are identified in their descriptions. A return value of zero indicates successful completion. Positive status codes are sent by the firmware; the Library generates negative status codes. See Section 3.6 for more information on status codes.

The intermediate-level processing allows you to control the host computer more closely. These functions perform the following:

- Initialize the communications port, set the timeout parameters and change the host computer communication rate.
- Request the low-level thread that clears the communication buffer, send and receive data stored in buffers and exit the low-level loop.
- Read from or write to the communications port.
- Set the user program to be synchronous or asynchronous control with the low-level thread.

For a complete list and description of commands, refer to Section 3.4.

## 2.3 High-Level Processing

The remaining functions are the high-level functions called in the user program to control the hand. The functions in this section have a common structure: they fill the output character buffer with the appropriate ASCII commands for the hand, and place a request to the low-level thread (using `ComRequest`). If a value is returned, the output from the hand is interpreted accordingly.

High-level processing allows you to control the hand via a set of standard functions. These functions perform the following:

- Initialize the communications port.
- Initialize the BarrettHand (resets parameters and aligns encoders and motors).
- Issue standard open and close commands for individual or multiple motors.
- Move individual or multiple motors to different positions.

- Terminate control of individual or multiple motors.
- Set or Get motor parameters.
- Load and save desired or default motor parameters to the EEPROM.
- Get BarrettHand temperature and communication rate.
- Send any set of characters to the BarrettHand.
- Allow control of the hand in RealTime mode.

The execution of all functions is affected by the settings of the serial port timeouts (via `ComSetTimeouts` in Section 3.4) and the three variables: `syncMode, requestTimeout,` and `pCallback.` See Section 3.5 for more information on these variables.

It is possible to use the high-level commands synchronously or asynchronously by setting the variable `syncMode`. During synchronous mode, the program will not regain control until the BarrettHand has finished executing the command or the timeout limit has been reached. During asynchronous mode, all user functions will return immediately with result `BHERR_PENDING`. You can check if the processing has been completed using `ComIsPending`, or wait for the processing to complete using `ComWaitForCompletion`. Except for RealTime mode functions, do not use asynchronous mode with functions that return values.

# 3 Using the C-Function Library

The C-Function Library consists of two files: *BHand.lib* and *BHand.h*. The Library file must be linked to the user program, and the header file must be included in the C++ code. The Library takes advantage of a number of C++ constructs that are incompatible with C; thus, a C++ compiler should be used. However, users can still write their programs in C style, since C++ is a superset of C.

The Library is compiled with Microsoft Visual C++ 6.0. The only functions used are standard Windows API functions (i.e. Microsoft Foundation Classes are not used), thus it should be straightforward to recompile the Library with a different Windows C++ compiler if necessary. Contact Barrett Technology if you will be using a different C++ compiler.

The *BHand.h* file defines the `class BHand`. Define an object of that class, and use it for all hand control. Make sure the object does not go out of scope (i.e. define it as a global variable or in the main function). If the `BHand` object goes out of scope and is lost, you will have to create and initialize a new one.

Here is an example of a simple program that initializes a hand on COM1, increases the baud rate, and closes all fingers (in practice you should check for status codes):

```
#include <BHand.h>
void main()
{
     BHand bh;             // Declare BHand object, bh
     char grasp[2] = "G";  // Defines grasp as "G"
     char all[1] = "";     // Defines all as ""

     bh.InitSoftware(1);   // initialize the Library, use COM 1
     bh.InitHand(all);     // Initialize the BarrettHand
     bh.Baud(19200);       // Change the baud rate to 19200bps
     bh.Close(grasp);      // Close the Grasp
}
```

The following sections will explain how to use all of the functions contained in the BarrettHand C-Function Library.

## 3.1 Installation

The C-Function Library, *BHand.lib*, and the header file, *BHand.h*, are needed to use the functions discussed in this manual. Run *setup.exe* on the CD labeled "C-Function Library" to install the Library files, example programs and a copy of the C-Function Library Manual. When you compile the software, make sure you link both the header file and the C-Function Library to the project.

## 3.2 User Functions – Supervisory Mode

### *3.2.1 Overview of Supervisory Mode*

Supervisory mode allows control of the hand at a high-level, allowing you to command individual or multiple motors to close, open, and move to specific positions. This set of commands is used for most grasping situations. This mode takes advantage of the supervisory capabilities of the Motorola 68HC11. The Motorola 68HC11 controls the Hand motion by supervising four HCTL-1100 motion-control microprocessors, one for each motor. If real-time control of the motor position, velocity, or strain is needed, use the new RealTime control mode described in Section 3.3.

In Supervisory mode, the Hand accepts a command from one of the functions in the C-Function Library and will not return control of the Hand until the command is finished processing or the C-Function Library calls SendControlC(). When the Hand completes its command, it may return a status code. See Section 3.6 for more detailed information on status codes.

### *3.2.2 Supervisory Mode Commands*

Note: *BHand.h* file defines the `class BHand`. Define an object of that class, and use it for all hand control. Make sure the object does not go out of scope (i.e. define it as either a global variable, or in the main function). All examples in this section assume an object named `bh` of class `BHand` and a status code of type `int` named `err` were previously defined.

**`Baud`**

*Syntax:* `int Baud( DWORD baud )`

*Arguments:* baud: The desired baud rate should be stored in this variable.

*Value:* baud: 600, 1200, 2400, 4800, 9600, 19200, 38400

*Example:*
```
// sets hand and serial port to 9600 baud
DWORD baud = 9600;
err = bh.Baud(baud);
```

*Purpose:* Changes the baud rate of both the associated Hand and the COM port on the host PC to the new value. The possible values are the standard baud rates up to 38400.

*Notes:* The baud rate of the hand is reset to 9600 by issuing the command `InitHand()`.

**`Buffer`**

*Syntax:* `const char* Buffer( void )`

*Arguments*: N/A

*Value:* N/A

*Example:*
```
// stores characters from the outbuf into ptrValue
const char* ptrValue;
ptrValue = bh.Buffer();
```

*Purpose:* Provides read-only access to the output buffer where hand commands waiting to be sent are stored.

*Notes:*

**Close**

*Syntax:* `int Close(char* motor)`

*Arguments:* motor: Specifies which motors will be closed.

*Value:*

| *Motor Parameter* | *Result* |
|---|---|
| 1 | Finger F1 |
| 2 | Finger F2 |
| 3 | Finger F3 |
| 4 or S | Spread Motion |
| G | Fingers F1, F2 and F3 |
| Empty String ( "" ) | All motors |

*Example:*

```
// closes grasp
char motor[4] = "123";
err = bh.Close(motor);
```

*Purpose:* Commands the selected motor(s) to move finger(s) in close direction with a velocity ramp-down to target limit, CT.

*Notes:* Finger(s) close until the joint stop(s) are reached, the close target is reached, or an obstacle is encountered.
The motor argument passed to the function needs to be a pointer to a string.

**Command**

*Syntax:* `int Command( char* send, char* receive )`

*Arguments:*

| | |
|---|---|
| send: | String to send to the BarrettHand. |
| receive: | Pointer to a buffer where the response will be stored. |

*Value:*

| | |
|---|---|
| send: | Any variation of letters and numbers. |
| receive: | Valid response to the command sent. |

*Example:*

```
// gets maximum close velocity of motor F3 and stores
// the resultant string value in receive
char command[10] = "3FGET MCV";
char receive[10];
err = bh.Command(command, &receive);
```

*Purpose:* Send an ASCII character string to the BarrettHand. The hand is expected to respond in Supervisory mode. If the `receive` buffer is supplied, the function will copy the hand response into the buffer. If not, you can obtain the hand response using the command `Response()`.

*Notes:* This function can be used to implement a simple terminal control. The command argument passed to the function needs to be a pointer to a string.

**Default**

*Syntax:* `int Default(char* motor)`

*Arguments:* motor: Specifies which motor's default parameters to load.

*Value:*

| *Motor Parameter* | *Result* |
|---|---|
| 1 | Finger F1 |
| 2 | Finger F2 |
| 3 | Finger F3 |
| 4 or S | Spread Motion |
| G | Fingers F1, F2 and F3 |
| Empty String ( "" ) | All motors |

*Example:*

```
// loads factory default parameters for the grasp
char motor[2] = "G";
err = bh.Default(motor);
```

*Purpose:* Loads factory default motor parameters from EEPROM into active parameter list.

*Notes:* This command only changes the active parameters, to write the parameters to EEPROM use `Save()`. The motor argument passed to the function needs to be a pointer to a string.

**Delay**

*Syntax:* `int Delay( DWORD msec );`

*Arguments:* msec: The desired delay should be stored in this variable.

*Value:* msec: 0 - 4.29E9 milliseconds

*Example:*

```
// Inserts a delay of 3 seconds
DWORD time = 3000;
err = bh.Delay(time);
```

*Purpose:* Insert a delay into sequence of commands.

*Notes:* None.

**Get**

*Syntax:* `int Get( char* motor, char* parameter, int* result )`

*Arguments:*

| | | |
|---|---|---|
| motor: | Specifies which motor's parameters to get. |
| parameter: | Specifies which motor parameter you want to get. |
| result: | The parameter values will be stored in this variable. |

*Value:*

| *Motor Parameter* | *Result* |
|---|---|
| 1 | Finger F1 |
| 2 | Finger F2 |
| 3 | Finger F3 |
| 4 or S | Spread Motion |
| G | Fingers F1, F2 and F3 |
| Empty String ( "" ) | All motors |

*Example:*

```
// gets the maximum close velocity for finger F1 and
// stores it in result
char motor[2] = "1";
char parameter[4] = "MCV";
int result;
err = bh.Get(motor, parameter, &result);
```

*Purpose:* Gets motor parameters.

*Notes:* See BH8-Series User Manual for a list of Motor Parameters and their functions. Verify that the size of the result variable can hold all values returned. For example, if you request the values for motors F1, F2 and F3, make sure you pass a pointer to an array with at least 3 valid locations. The motor and parameter arguments passed to the function need to be pointers to strings.

**GoToDefault**

*Syntax:* `int GoToDefault( char* motor );`

*Arguments:* motor: Specifies which motors will be moved.

*Value:*

| *Motor Parameter* | *Result* |
|---|---|
| 1 | Finger F1 |
| 2 | Finger F2 |
| 3 | Finger F3 |
| 4 or S | Spread Motion |
| G | Fingers F1, F2 and F3 |
| Empty String ( "" ) | All motors |

*Example:*

```
// move grasp to default positions
char motor[2] = "G";
err = bh.GoToDefault(motor);
```

*Purpose:* Moves all motors to default positions defined by the default parameter DP.

*Notes:* The motor argument passed to the function needs to be a pointer to a string.

**GoToDifferentPositions**

*Syntax:* `int GoToDifferentPositions( int value1, int value2, int value3, int value4 )`

*Arguments:* value1,2,3,4: Specifies the encoder position for each motor respectively.

*Value:* value1,2,3,4: 0 - 20,000

*Example:*
```
// moves finger F1 to 2000, finger F2 to 3000, finger F3
// to 4000 and Spread to 1000
err = bh.GoToDifferentPositions(2000, 3000, 4000, 1000);
```

*Purpose:* Moves all motors to specified encoder positions.

*Notes:*

**GoToHome**

*Syntax:* `int GoToHome(void)`

*Arguments:* N/A

Value: N/A

*Example:*
```
// moves all motors to the home position
err = bh.GoToHome();
```

*Purpose:* Moves all motors to the home position, driving all fingers and the spread to their full open position. See the BarrettHand User Manual for more information on the home position.

*Notes:*

**GoToPosition**

*Syntax:* `int GoToPosition( char* motor, int value )`

*Arguments:* motor: Specifies which motors will be closed.
value: Specifies the encoder position to be moved to.

*Value:*

| *Motor Parameter* | *Result* |
|---|---|
| 1 | Finger F1 |
| 2 | Finger F2 |
| 3 | Finger F3 |
| 4 or S | Spread Motion |
| G | Fingers F1, F2 and F3 |
| Empty String ( "" ) | All motors |

value: 0 - 20,000 encoder counts

*Example:*

```
// moves finger F3 to position 10000
char motor[2] = "3";
err = bh.GoToPosition(motor, 10000);
```

*Purpose:* Moves motors to specified encoder position.

*Notes:*

**InitHand**

*Syntax:* `int InitHand(char* motor)`

*Arguments:* motor: Specifies motors to initialize.

*Value:*

| *Motor Parameter* | *Result* |
|---|---|
| 1 | Finger F1 |
| 2 | Finger F2 |
| 3 | Finger F3 |
| 4 or S | Spread Motion |
| G | Fingers F1, F2 and F3 |
| Empty String ( "" ) | All motors |

*Example:*

```
// initializes all Finger motors
char motor[2] = "G";
err = bh.InitHand(motor);
```

*Purpose:* Determines encoder and motor alignment for commutation, moves all fingers and spread to open positions and resets the baud rate to 9600.

*Notes:* `InitHand()` needs to be called after the hand has been reset. This command must be run before any other motor commands, once the hand is turned on.
The motor argument passed to the function needs to be a pointer to a string.

**InitSoftware**

*Syntax:* `int InitSoftware(int port, int priority = THREAD_PRIORITY_TIME_CRITICAL)`

*Arguments:*
port: Specifies the serial port to use.
priority: Specifies the priority of the low-level thread. If not specified, the value is set to the highest priority.

*Value:*
port: can be set to any valid serial port on the host computer
priority: THREAD_PRIORITY_IDLE
THREAD_PRIORITY_LOWEST
THREAD_PRIORITY_BELOW_NORMAL
THREAD_PRIORITY_NORMAL
THREAD_PRIORITY_ABOVE_NORMAL
THREAD_PRIORITY_HIGHEST
THREAD_PRIORITY_TIME_CRITICAL

*Example:*
```
// initializes the software for communications port 1
// initializes the low-level thread to be
// THREAD_PRIORITY_TIME_CRITICAL
err = bh.InitSoftware(1, THREAD_PRIORITY_TIME_CRITICAL);
```

*Purpose:* This function resets all internal variables to their defaults:
```
syncMode = BHMODE_SYNC
pCallback  = NULL
requestTimeout  = INFINITE
```
It then opens the communications port, sets the low-level thread priority, and resets the hand (setting the baud rate to 9600)

*Notes:* User must call `InitHand()` after resetting the hand. Barrett recommends that the priority not be changed unless you are familiar with thread priorities.

**Load**

*Syntax:* `int Load(char* motor)`

*Arguments:* motor: Specifies which motor's parameters to load.

*Value:*

| *Motor Parameter* | *Result* |
|---|---|
| 1 | Finger F1 |
| 2 | Finger F2 |
| 3 | Finger F3 |
| 4 or S | Spread Motion |
| G | Fingers F1, F2 and F3 |
| Empty String ( "" ) | All motors |

*Example:*

```
// loads previously saved parameters for the grasp
char motor[2] = "G";
err = bh.Load(motor);
```

*Purpose:* Loads the saved motor parameters from EEPROM into active parameter list.

*Notes:* The motor argument passed to the function needs to be a pointer to a string. All of the settable firmware parameters will be loaded into RAM.

**Open**

*Syntax:* `int Open(char* motor)`

*Arguments:* motor: Specifies which motors will be opened.

*Value:*

| *Motor Parameter* | *Result* |
|---|---|
| 1 | Finger F1 |
| 2 | Finger F2 |
| 3 | Finger F3 |
| 4 or S | Spread Motion |
| G | Fingers F1, F2 and F3 |
| Empty String ( "" ) | All motors |

*Example:*

```
// opens the spread
char motor[2] = "S";
err = bh.Open(motor);
```

*Purpose:* Commands the selected motor(s) to move finger(s) in open direction with a velocity ramp-down at target limit, OT.

*Notes:* Finger(s) open until the open target is reached or an obstacle is encountered.
The motor argument passed to the function needs to be a pointer to a string.

**PGet**

*Syntax:* `int PGet( char* parameter )`

*Arguments:* parameter: Specifies which Global parameter will be read.

*Example:* // Get over temperature fault value)

```
char parameter[6] = "OTEMP";
int result;
result = bh.PGet( parameter );
```

*Notes:* See BH8-Series User Manual for a list of Global Parameters and their functions.

**PSet**

*Syntax:* `int PSet( char* parameter, int value )`

*Arguments:* parameter: Specifies which Global parameter will be set.
value: Specifies the desired value of the parameter.

*Example:* // set over temperature fault to 585 (58.5 degrees Celsius)

```
char parameter[6] = "OTEMP";
err = bh.PSet( parameter, 585);
```

*Notes:* See BH8-Series User Manual for a list of Global Parameters and their functions.

**Reset**

*Syntax:* `int Reset(void)`

*Arguments:* N/A

*Value:* N/A

*Example:*

```
// resets the hand
err = bh.Reset();
```

*Purpose:* Resets the firmware loop in the BarrettHand and sets the baud rate to 9600.

*Notes:* After resetting the BarrettHand you will need to call `InitHand()` before issuing any motion commands.

**Response**

*Syntax:* `const char* Response( void )`

*Arguments*: N/A

*Value:* N/A

*Example:*
```
//stores characters from the input buffer into ptrValue
const char* ptrValue;
ptrValue = bh.Response();
```

*Purpose:* Provides read-only access to the input buffer where hand responses are stored.

*Notes:* Leading and trailing white space characters are eliminated, as well as the "=>" prompt at the end of the string. If the Library is used in asynchronous mode, wait for request completion before reading the response buffer.

**Save**

*Syntax:* `int Save( char* motor )`

*Arguments:* motor: Specifies which motor's parameters to save.

*Value:*

| *Motor Parameter* | *Result* |
|---|---|
| 1 | Finger F1 |
| 2 | Finger F2 |
| 3 | Finger F3 |
| 4 or S | Spread Motion |
| G | Fingers F1, F2 and F3 |
| Empty String ( "" ) | All motors |

*Example:*
```
// saves the grasp motor parameters
char motor[2] = "G";
err = bh.Save(motor);
```

*Purpose:* Saves present motor parameters from the active parameter list to EEPROM. These values can be loaded later. Storing the values in EEPROM allows you to reset the BarrettHand and retain preferred motor parameters.

*Notes:* The parameters can be recalled into the active parameter list by using the function `Load()` The motor argument passed to the function needs to be a pointer to a string. However, this command should not be performed more than 5,000 times or the Hand electronics may need repair.

**SendControlC**

*Syntax:* `int SendControlC( void )`

*Arguments:* N/A

*Value:* N/A

*Example:*
```
//terminates Command
err = bh.SendControlC();
```

*Purpose:* Sends a 'Control C' character to the BarrettHand. Terminates control of the hand.

*Notes:*

**Set**

*Syntax:* `int Set( char* motor, char* parameter, int value )`

*Arguments:* motor: Specifies which motor's parameters to set.
parameter: Specifies which motor parameter will be set.
value: Specifies the desired value of the parameter.

*Value:*

| *Motor Parameter* | *Result* |
|---|---|
| 1 | Finger F1 |
| 2 | Finger F2 |
| 3 | Finger F3 |
| 4 or S | Spread Motion |
| G | Fingers F1, F2 and F3 |
| Empty String ( "" ) | All motors |

*Example:*
```
// set finger F1 maximum close velocity to 20
char motor[2] = "1";
char parameter[4] = "MCV";
err = bh.Set(motor, parameter, 20);
```

*Purpose:* Sets motor parameters.

*Notes:* See BH8-Series User Manual for a list of Motor Parameters and their functions. The motor and parameter arguments passed to the function need to be pointers to strings.

**StepClose**

*Syntax:* `int StepClose( char* motor, int value )`

*Arguments:* motor: Specifies which motors will be closed.
value: Specifies the size of the incremental close.

*Value:*

| *Motor Parameter* | *Result* |
|---|---|
| 1 | Finger F1 |
| 2 | Finger F2 |
| 3 | Finger F3 |
| 4 or S | Spread Motion |
| G | Fingers F1, F2 and F3 |
| Empty String ( "" ) | All motors |

value: 0 - 20,000 encoder counts

*Example:*

```
// step close finger F2 1500 encoder counts
char motor[2] = "2";
err = bh.StepClose(motor, 1500);
```

*Purpose:* Incrementally closes the specified motors. The step amount can either be specified or the default size will be used. The parameter `DS` contains the default increment size.

*Notes:*

**StepOpen**

*Syntax:* `int StepOpen( char* motor, int value )`

*Arguments:* motor: Specifies which motors will be opened.
value: Specifies the size of the incremental open.

*Value:*

| *Motor Parameter* | *Result* |
|---|---|
| 1 | Finger F1 |
| 2 | Finger F2 |
| 3 | Finger F3 |
| 4 or S | Spread Motion |
| G | Fingers F1, F2 and F3 |
| Empty String ( "" ) | All motors |

value: 0 - 20,000 encoder counts

*Example:*

```
// step open the grasp 2000 encoder counts
char motor[2] = "G";
err = bh.StepOpen(motor, 2000);
```

*Purpose:* Incrementally opens the specified motors. The step amount can either be specified or the default value will be used. The parameter `DS` contains the default increment size.

*Notes:*

**StopMotor**

*Syntax:* `int StopMotor( char* motor )`

*Arguments:* motor: Specifies which motors will be terminated.

*Value:*

| *Motor Parameter* | *Result* |
|---|---|
| 1 | Finger F1 |
| 2 | Finger F2 |
| 3 | Finger F3 |
| 4 or S | Spread Motion |
| G | Fingers F1, F2 and F3 |
| Empty String ( "" ) | All motors |

*Example:*

```
// stops actuating the spread motor
char motor[2] = "S";
err = bh.StopMotor(motor);
```

*Purpose:* Stops actuating motors specified.

*Notes:* Because the fingers are not backdrivable, the finger motors are actually terminated after finished moving. The spread motion is backdrivable and is therefore commanded to hold position even after the desired position is attained. Use the `StopMotor()` command, when possible, to reduce the amount of heat generated by the Hand.
The motor argument passed to the function needs to be a pointer to a string.

**TorqueClose**

*Syntax:* `int TorqueClose(char* motor)`

*Arguments:* motor: Specifies which motors will be closed with torque control.

*Value:*

| *Motor Parameter* | *Result* |
|---|---|
| 1 | Finger F1 |
| 2 | Finger F2 |
| 3 | Finger F3 |
| 4 or S | Spread Motion |
| G | Fingers F1, F2 and F3 |
| Empty String ( "" ) | All motors |

*Example:*

```
// closes grasp with torque control
char motor[4] = "123";
err = bh.TorqueClose(motor);
```

*Purpose:* Commands velocity of selected motor(s) in the direction that closes the finger(s) with control of motor torque at stall.

*Notes:*

**Torqueopen**

*Syntax:* `int Torqueopen(char* motor)`

*Arguments:* motor: Specifies which motors will be closed with torque control.

*Value:*

| *Motor Parameter* | *Result* |
|---|---|
| 1 | Finger F1 |
| 2 | Finger F2 |
| 3 | Finger F3 |
| 4 or S | Spread Motion |
| G | Fingers F1, F2 and F3 |
| Empty String ( "" ) | All motors |

*Example:*
```
// opens grasp with torque control
      char motor[4] = "123";
      err = bh.TorqueOpen(motor);
```

*Purpose:* Commands velocity of selected motor(s) in the direction that opens the finger(s) with control of motor torque at stall.

*Notes:*

**Temperature**

*Syntax:* `int Temperature(int* result)`

*Arguments:* result: The temperature will be stored in this variable.

*Value:* result: The result will be returned in degrees Celsius.

*Example:*
```
// stores the temperature in result
int result;
err = bh.Temperature(&result);
```

*Purpose:* Returns the temperature from the temperature sensor on the BarrettHand.

*Notes:*

## 3.3 User Functions – RealTime Mode

### *3.3.1 Overview of RealTime Mode*

RealTime mode allows users to write custom control laws that can be computed by a powerful host processor. This mode allows the host computer to send low-level commands to, and receive feedback from, the Hand in real time.

RealTime mode is implemented in the following sequence:

1. Prepare RealTime mode:
    a. Use RTSetFlags() to define the subset of information to be sent by the host PC and by the Hand. Custom communication-packet blocks are then optimized for fastest possible bandwidth.

   b. Call RTStart() to switch from Supervisory mode to RealTime mode. (Note that the Hand always begins in Supervisory mode upon powering up.)
2. Command RealTime mode:
   a. Write command parameters to a host buffer with RTSet commands.
   b. Use RTUpdate() to trigger the exchange of data already specified by RTSetFlags().
   c. Read feedback parameters from the host buffer with RTGet commands.
3. Exit RealTime mode by using RTAbort().

RealTime parameters are either flags or coefficients. RealTime flags determine what data will be sent and received continuously from the hand during RealTime control. RealTime coefficients affect commands that are used during RealTime control but are set once and not continuously transmitted. You can specify different blocks of information for each motor by setting RealTime flags for each motor. You can also set different values for RealTime variables for each motor. The possible control data that can be sent are given in the RealTime Control Parameter Tables.

### *3.3.2 RealTime Mode Commands*

Note: *BHand.h* file defines the `class BHand`. Define an object of that class, and use it for all hand control. Make sure the object does not go out of scope (i.e. define it as either a global variable, or in the main function). All examples in this section are assuming an object named `bh` of class `BHand` was previously defined.

**`RTAbort`**

*Syntax:* `int RTAbort( void )`

*Arguments:* N/A

*Value:* N/A

*Example:*
```
//Ends RealTime control
err = bh.RTAbort();
```

*Purpose:* Ends RealTime mode by sending the Hand a Cntl-C and returns to Supervisory mode.

*Notes:* N/A

**RTGetDeltaPos**

*Syntax:* `char RTGetDeltaPos(char motor)`

*Arguments:* motor: Determines which motor's delta position will be retrieved.

*Value:*

| *Motor Parameter* | *Result* |
|---|---|
| 1 | Finger F1 |
| 2 | Finger F2 |
| 3 | Finger F3 |
| 4 or S | Spread Motion |

*Example:*

```
//Gets delta position for motor F1
char DeltaPosition;
DeltaPosition = bh.RTGetDeltaPos('1');
```

*Purpose:* Gets RealTime delta position feedback for the desired motor.

*Notes:* Only one motor value can be retrieved at a time.
Delta position is the change in position from the last reported position and is limited to one signed byte. The present position is read and compared to the last reported position. The difference is divided by the RealTime variable LFDPC, clipped to a single signed byte, and then sent to the host. The value sent to the host should then be multiplied by LFDPC and added to the last reported position.

Example (with LFDPC set to 2): What will delta position feedback look like if last reported position was 1500 and the position jumps to 2000? The first feedback block will include the delta position value 127. This value should be multiplied by LFDPC on the host machine resulting in 254. The hand will internally update the reported position to 1754. The next feedback block will include the delta position 123, which should be multiplied by LFDPC resulting in 246. The reported position will be updated to 2000. Subsequent feedback blocks will include the delta position value 0 (until the next position change).

**RTGetPosition**

*Syntax:* `int RTGetPosition(char motor)`

*Arguments:* motor: Determines which motor's absolute position will be retrieved.

*Value:*

| *Motor Parameter* | *Result* |
|---|---|
| 1 | Finger F1 |
| 2 | Finger F2 |
| 3 | Finger F3 |
| 4 or S | Spread Motion |

*Example:*

```
//Gets absolute position for motor F1
int Position;
Position = bh.RTGetPosition('1');
```

*Purpose:* Gets RealTime absolute position feedback for the desired motor.

*Notes:* Only one motor value can be retrieved at a time.

**RTGetStrain**

*Syntax:* `unsigned char RTGetStrain(char motor)`

*Arguments:* motor: Determines which finger strain gage values will be retrieved.

*Value:*

| *Motor Parameter* | *Result* |
|---|---|
| 1 | Finger F1 |
| 2 | Finger F2 |
| 3 | Finger F3 |
| 4 or S | Spread Motion |

*Example:*

```
//Gets strain gage values for motor F3
unsigned char Strain;
Strain = bh.RTGetStrain('3');
```

*Purpose:* Gets RealTime strain gage feedback value for the desired motor.

*Notes:* Only one motor value can be retrieved at a time.

**RTGetVelocity**

*Syntax:* `char RTGetVelocity(char motor)`

*Arguments:* motor: Determines which motor velocities will be retrieved.

*Value:*

| *Motor Parameter* | *Result* |
|---|---|
| 1 | Finger F1 |
| 2 | Finger F2 |
| 3 | Finger F3 |
| 4 or S | Spread Motion |

*Example:*

```
//Get actual velocity of motor F1
char Velocity;
Velocity = bh.RTGetVelocity('1');
```

*Purpose:* Gets RealTime actual velocity value for the desired motor.

*Notes:* Only one motor value can be retrieved at a time.

**RTGetTemp**

*Syntax:* `int RTGetTemp ( void )`

*Arguments:* N/A

*Example:*

```
//Get temperature
Int handTemperature;
handTemperature = RTGetTemp();
```

*Purpose:* Gets RealTime temperature value.

*Notes:*

**RTSetFlags**

*Syntax:*
```
int RTSetFlags( char* motor, bool LCV, int LCVC, bool
LCPG, bool LFV, int LFVC, bool LFS, bool LFAP, bool LFDP,
int LFDPC, int LFT, bool LFDPD )
```

*Arguments:*

| | |
|---|---|
| motor: | Determines which motor parameters will be set. |
| parameters: | Loop control or loop feedback parameters. |

*Value:*

| *Motor Parameter* | *Result* |
|---|---|
| 1 | Finger F1 |
| 2 | Finger F2 |
| 3 | Finger F3 |
| 4 or S | Spread Motion |
| G | Fingers F1, F2 and F3 |
| Empty String ( "" ) | All motors |

*Example:*
```
//Prepares flags and variables to send control velocity
//and receive absolute position and temperature to/from
motor F2
char motor[2] = "2";
err = bh.RTSetFlags(motor, TRUE, 1, FALSE, FALSE, 1,
FALSE, TRUE, FALSE, 1, TRUE, FALSE);
```

*Purpose:* Sets the eleven parameters relevant for RealTime mode, for the specified motors. See BH8-Series User Manual for a list of relevant RealTime Parameters and their functions.

*Notes:* This function is provided for convenience, the same effect can be achieved with multiple calls to the `Set()` function in Section 3.2.2. However, RTSetFlags() can only define some of the parameters that can be defined with Set().
The motor argument passed to the function needs to be a pointer to a string.

**RTSetGain**

*Syntax:* `int RTSetGain( char motor, int gain )`

*Arguments:* motor: Determines which motor gain will be set.
gain: Desired proportional gain for the selected motors.

*Value:*

| *Motor Parameter* | *Result* |
|---|---|
| 1 | Finger F1 |
| 2 | Finger F2 |
| 3 | Finger F3 |
| 4 or S | Spread Motion |

gain: 0 - 255

*Example:*

```
//Set gain parameters of motors F1 and F2 to 150
err = bh.RTSetGain('1', 150);
err = bh.RTSetGain('2', 150);
```

*Purpose:* Sets RealTime proportional gain parameter for the desired motor.

*Notes:* Only one motor can be set at a time.
The gains for the motors will not actually be set until `RTUpdate()` is called.
In RealTime control, the motors are controlled using a proportional velocity mode. The proportional gain affects the motor command according to the Velocity Control equations in Section 3.2.1.

**RTSetVelocity**

*Syntax:* `int RTSetVelocity( char motor, int velocity )`

*Arguments:* motor: Determines which motor velocity will be set.
velocity: Desired control velocity for the selected motors.

*Value:*

| *Motor Parameter* | *Result* |
|---|---|
| 1 | Finger F1 |
| 2 | Finger F2 |
| 3 | Finger F3 |
| 4 or S | Spread Motion |

velocity: -128 to 127

*Example:*

```
// Set control velocity parameters of motors F1, F2, and
// F3 to 50
err = bh.RTSetVelocity('1', 50);
err = bh.RTSetVelocity('2', 50);
err = bh.RTSetVelocity('3', 50);
```

*Purpose:* Sets RealTime control velocity parameter for the desired motor.

*Notes:* The motors will not actually be set to this control velocity until `RTUpdate()` is called. Only one motor can be set at a time.

**RTStart**

*Syntax:* `int RTStart( char* motor )`

*Arguments:* motor: determines which motors will be controlled in RealTime

*Value:*

| *Motor Parameter* | *Result* |
|---|---|
| 1 | Finger F1 |
| 2 | Finger F2 |
| 3 | Finger F3 |
| 4 or S | Spread Motion |
| G | Fingers F1, F2 and F3 |
| Empty String ( "" ) | All motors |

*Example:*

```
//Enter motor F2 into RealTime control
char motor[2] = "2";
err = bh.RTStart(motor);
```

*Purpose:* Call this function when all parameters have been set and you are ready to enter RealTime control.

*Notes:* The motor argument passed to the function needs to be a pointer to a string.

**RTUpdate**

*Syntax:* `int RTUpdate( bool control, bool feedback )`

*Arguments:* control: Indicates if control data should be sent.
feedback: Indicates if feedback data should be received.

*Value:* control: TRUE or FALSE
feedback: TRUE or FALSE

*Example:*

```
//Set Velocity to 30 and read position for motor F2, stop
//when position > 3000

char motor[2]="2";
char parameter[2]="P";
int pos[1];

err = bh.RTSetFlags(motor, TRUE, 1, FALSE, FALSE, 1,
FALSE, TRUE, FALSE, 1, TRUE, FALSE);
err = bh.Get(motor, parameter, pos);
err = bh.RTStart(motor);

while (pos[1] < 3000)
{
     err = bh.RTSetVelocity('2', 30);
     err = bh.RTUpdate(TRUE, TRUE);
     pos = bh.RTGetPosition('2');
}
```

*Purpose:* This command is used to trigger the sending and receiving of data between the host PC and the Hand.

*Notes:*

### 3.4 Serial Communication Functions

The serial communication functions allow you to communicate with the hand without ever using the high-level Supervisory or RealTime commands. These functions allow you to take complete advantage of the Library functionality.

Note: *BHand.h* file defines the `class BHand`. Define an object of that class, and use it for all hand control. Make sure the object does not go out of scope (i.e. define it as either a global variable, or in the main function). All examples in this section assume an object named `bh` of class `BHand` and a status code of type `int` named `err` were previously defined.

#### `ComClear`

*Syntax:* `bool ComClear( void )`

*Arguments:* N/A

*Value:* Returns TRUE if the clear operation is successful, FALSE if an error occurs.

*Example:*
```
// clears characters from the input and output buffers
isCleared = bh.ComClear();
```

*Purpose:* This function clears all characters from the send and receive buffers of the serial port.

*Notes:* None.

**ComInitialize**

*Syntax:* `int ComInitialize( unsigned char comport, int priority )`

*Arguments:* comport: Specifies communications port.
priority: Specifies low-level thread priority.

*Value:* comport: Any valid communications port on host computer.
priority: THREAD_PRIORITY_IDLE
THREAD_PRIORITY_LOWEST
THREAD_PRIORITY_BELOW_NORMAL
THREAD_PRIORITY_NORMAL
THREAD_PRIORITY_ABOVE_NORMAL
THREAD_PRIORITY_HIGHEST
THREAD_PRIORITY_TIME_CRITICAL

*Example:*
```
// initializes communications port 2 with priority set to
// THREAD_PRIORITY_TIME_CRITICAL
err = bh.ComInitialize('2', THREAD_PRIORITY_TIME_CRITICAL);
```

*Purpose:* This function opens the serial port with the specified number (1, 2, …) , sets the baud rate to 9600, serial port timeouts to default values and initializes the low-level thread to the desired priority.

*Notes:* The default thread priority used in the Library is THREAD_PRIORITY_TIME_CRITICAL, the highest possible. For more information on threads and priorities, read the Windows API documentation.

**ComIsPending**

*Syntax:* `bool ComIsPending( void )`

*Arguments:* N/A

*Value:* Returns TRUE if a request to the low-level thread is still pending (or being processed), and FALSE if no request is pending.

*Example:*
```
// checks to see if there is a process still pending
isPending = bh.ComIsPending(void);
```

*Purpose:* Determines if a request to the low-level thread is still pending or being processed.

*Notes:* None.

**ComRead**

*Syntax:* `int ComRead( char* buffer, int size )`

*Arguments:* buffer: Characters read from serial port are put into the location pointed to by `*buffer.`
size: Represents the number of characters to be read into the buffer.

*Value:* buffer: ASCII characters sent from the hand.
size: Number of characters returned from hand.

*Example:*
```
// reads one character from input buffer into inputbuffer
err = bh.ComRead(&inputbuffer, 1);
```

*Purpose:* Reads `size` characters from the serial port into the character buffer pointed to by `*buffer`.

*Notes:* This function is affected by the present settings of the serial port timeout parameters. The low-level thread uses this function to read data, therefore it is executed with high priority.

**ComRequest**

*Syntax*: `int ComRequest( int requestNumber )`

*Arguments:* request number: Type of request from low-level thread.

*Value:* request number:

| | |
|---|---|
| BHREQ_EXIT: | exit the low-level loop (called before the Library is closed) |
| BHREQ_REALTIME: | RealTime command: expect response starting with '*' |
| BHREQ_SUPERVISE: | Supervise command: expect response ending with "=>" |
| BHREQ_CLEAR: | clear com port buffers |

*Example:*
```
// Request to clear the com port buffers
err = bh.ComRequest(BHREQ_CLEAR);
```

*Purpose:* This function places a request to the low-level thread. This can be used to receive information, clear the communication port buffers and exit the low-level thread.

*Notes:* The response to `BHREQ_REALTIME` and `BHREQ_SUPERVISE` can be retrieved by using the command `ComRead()`.

**ComSetBaudrate**

*Syntax:* `int ComSetBaudrate( DWORD baud )`

*Arguments:* baud: Rate of communication for selected serial port.

*Value:* baud: 600, 1200, 2400, 4800, 9600, 19200, 38400

*Example:*
```
// sets serial communication speed for the host computer
// to 9600 baud
err = bh.ComSetBaudrate(9600);
```

*Purpose:* Sets the selected serial port speed to the specified baud rate.

*Notes:* Consider using the Baud function instead. Calling the ComSetBaudrate function has to be accompanied by appropriate commands to the BarrettHand hardware, so that its baud rate changes accordingly. In general, if the Library and the firmware decide to communicate at different baud rates, all functions will return errors. To avoid that situation, always make sure the hand is reset before starting the software, and do not modify the software baud rate and the hand baud rate separately. The high-level function `Baud()` makes all necessary modifications.

**ComSetTimeouts**

*Syntax:*
```
int ComSetTimeouts( DWORD readInterval, DWORD
readMultiplier, DWORD readConstant, DWORD writeMultiplier,
DWORD writeConstant )
```

*Arguments:*

| | |
|---|---|
| readInterval: | Maximum interval between receiving two consecutive characters. |
| readMuliplier: | Average time per character. |
| readConstant: | Constant time for entire transaction. |
| writeMultiplier: | Average time per character. |
| writeConstant: | Constant time for the entire transaction. |

*Value:* All timeout parameters are in milliseconds. A value of zero (0) disables the respective timeout, essentially making the timeout infinite.

*Example:*
```
// sets
// read interval to 0
// read multiplier to 100
// read constant to 15000
// write multiplier to 100
// write constant to 5000
err = bh.ComSetTimeouts(0, 100, 15000, 100, 5000);
```

*Purpose:* Sets the serial port timeout parameters to the specified values.

*Notes:* It is recommended to remain at the default values unless shorter or longer timeouts are required in your application. Even at the slowest possible baud rate of 600baud, the communications will not timeout under normal circumstances. The Com`Read()` function returns a timeout error if any between-character interval is too large, or the total amount of time exceeds (readConstant + NumChars*readMultiplier). The write timeout interval is calculated by adding the writeConstant to the product of the writeMulitplier and the number of bytes to be written (writeConstant + NumChars*writeMultiplier).

**ComWaitForCompletion**

*Syntax:* `int ComWaitForCompletion( DWORD timeout )`

*Arguments:* timeout: maximum length of time to block the user program.

*Value:* timeout: 0 to INFINITE

*Example:*
```
// waits for low-level thread to finish processing the
// present request.  The function will never time out
err = bh.ComWaitForCompletion(INFINITE);
```

*Purpose:* Blocks execution of the user program until the low-level thread is finished processing the present request or the timeout value has been exceeded.

*Notes:* None.

**`ComWrite`**

*Syntax:* `int ComWrite( char* buffer, int size )`

*Arguments:*
- buffer: Characters in the location pointed to by `*buffer` are sent to the serial port.
- size: Represents the number of characters to be sent from the buffer.

*Value:*
- buffer: Any ASCII characters can be sent.
- size: The number of characters to be sent.

*Example:*

```
// sends O to hand, resulting in all motors opening
char buffer[2] = "O";
err = bh.ComWrite(buffer, 1);
```

*Purpose:* Writes `size` characters to the serial port from the character buffer pointed to by `*buffer`.

*Notes:* This function is affected by the present settings of the serial port timeout parameters. The low-level thread uses this function to read data, therefore it is executed with high priority.

### 3.5 C-Function Library Variables

This section defines the different variables within the C-Function Library.

*Variable:* **`pCallback`**

*Type:* `BHCallback`

*Purpose:* This variable, if different from NULL, is a pointer to a function that will be called right before the low-level thread signals the user program that processing has finished.

*Values:* any function

*Default:* NULL

*Notes:* The function will be executed with high priority so it should not be computationally intensive. The callback function type is: `typedef void (*BHCallback)( class BHand* )`. Following is an example:

```
void RealTimeCallBackFunction(BHand* noptr)
{
      /* Insert code to be executed here */
}
```

In your `main()` function use the following assignment:

```
pCallBack = RealTimeCallBackFunction;
```

*Variable:* **`requestTimeout`**

*Type:* DWORD

*Purpose:* This variable specifies the timeout interval (in milliseconds) used in synchronous mode.

*Values:* positive integers

*Default:* INFINITE

*Notes:* INFINITE specifies that the user program does not resume until the low-level thread is finished processing the present request.

*Variable:* **`syncMode`**

*Type:* integer

*Purpose:* This variable determines whether/how the user program waits for the low-level thread to complete the request before it continues.

*Values:*

| | |
|---|---|
| BHMODE_SYNCH: | User program waits for completion of low-level thread. |
| BHMODE_ASYNCHNOW: | Try to send request now (error if another request is being processed), do not wait for completion. |
| BHMODE_ASYNCHWAIT: | Wait for completion of previous request, then send request and return immediately |
| BHMODE_RETURN: | Send only requests for parameters (Temperature or Get commands), disregard all other requests, run the callback function and return. |

*Default:* BHMODE_SYNCH

*Notes:* Setting the variable to asynchronous mode allows you to continue program execution while the request is still being processed.
Do not use asynchronous mode when a result is to be returned.

### 3.6 Status and Error Codes

For all functions in the C-Function Library that return an integer, the value is determined according to the following rule:

1. If an error in the use of the C-Function Library itself is detected, such as inability to read from the PC's COM port, a negative integer is returned according to the errors listed in Table 2 (also listed in the BHand.h file);
2. Else, if the BarrettHand Firmware reports one or more status-code integers according to the Table for Status Codes in the User Manual, the sum of those integers is returned;
3. Else zero "0" is returned, indicating successful completion of the function.

**Table 2 – C-Function Library Error Codes**

| *Library Status #* | *Name* | *Description* |
|---|---|---|
| 0 | N/A | Successful completion of function |
| -1 | BHERR_BHANDEXISTS | Attempt to initialize a second Bhand object on a com port already running a Bhand object. |
| -2 | BHERR_OPENCOMMPORT | Error opening the specified com port |
| -3 | BHERR_GETCOMMSTATE | Could not read the state of the com port |
| -4 | BHERR_SETCOMMSTATE | Could not set the state of the com port |
| -5 | BHERR_SETCOMMTIMEOUT | Could not set the com port timeout parameters |
| -6 | BHERR_THREADSTART | Could not start the low-level thread |
| -7 | BHERR_THREADPRIORITY | Error setting the thread priority |
| -8 | BHERR_WRITECOM | Error writing to the com port (including timeout) |
| -9 | BHERR_READCOM | Error reading from the com port (including timeout) |
| -10 | BHERR_BADRESPONSE | Hand responded with an incorrect sequence of characters |
| -11 | BHERR_CLEARBUFFER | Could not clear com port buffers |
| -12 | BHERR_TIMEOUT | Request to low-level thread timed out |
| -13 | BHERR_NOTCOMPLETED | Previous request not completed (in ASYNCNOW) |
| -14 | BHERR_PENDING | Request is still being processed (normal in ASYNC mode) |
| -15 | BHERR_NOTINITIALIZED | A BHand object is not initialized (with InitSoftware) |
| -16 | BHERR_BADPARAMETER | The parameter code passed to Get is invalid |
| -17 | BHERR_LONGSTRING | Send or receive string exceeds BH_MAXCHAR |
| -18 | BHERR_OUTOFRANGE | The parameter is not within a valid range |
| -19 | BHERR_MOTORINACTIVE | The motor is not activated |
| -20 | BHERR_PORTOUTOFRANGE | The requested serial port is out of range |

### 3.7 Version 1.0 C-Function Library Compatibility

This section describes source code compatibility with the previous Library, Version 1.0. It is possible to recompile existing programs with these changes:

- Remove `#include "stdc.h"` and `#include "serial.h"` if they exist.
- Add `#include "BHand.h"`.
- Change the extension of the program file to be **.cpp*.

All functions from version 1.0 have been implemented to call the corresponding functions in version 4.3. `InitSoftware` has been modified to silently allocate a `BHand` object (pointed to by the global variable `BHand* _pBh`). All other functions use that object to translate calls to the hand. Calling a Version 1.0 function without first calling `InitSoftware,` returns with error `BHERR_NOTINITIALIZED`.

The functions in this group are:

```
int   Open( char motor );
int   Close( char motor );
int   StepOpen( char motor, int value );
int   StepClose( char motor, int value );
int   GoToPosition( char motor, int value );
int   GoToDifferentPositions(int value1, int value2, int value3, int
      value4);
int   GoToHome( void );
int   StopMotor( char motor );
int   Set( char motor, char parameter, int value );
int   Get( char motor, char parameter, int* result );
int   Save( char motor );
int   Load( char motor );
int   Default( char motor );
int   Temperature( int* result );
int   InitSoftware( void );
int   InitHand( void );
```

They behave exactly as in Version 1.0 of the Library. Here we only describe the differences between these functions and their equivalents in the 4.3 Library:

- The `Get` and `Set` functions accept a character flag for the parameter, which is internally translated into a string holding the new parameter name (some of the parameter flags have changed with the new firmware). The possible flags are the same as in Version 1.0.
- The `InitSoftware` function does not accept any parameters – the Library is always initialized to use COM1, highest low-level thread priority, baud rate 9600, default communication port timeouts, infinite request timeout, synchronous mode and no callback function.

**Note**: Barrett Technology does not recommend mixing version 1.0 and 4.3 function calls. If you call `InitSoftware()` initializing the 1.0 Library commands, you should use the Version 1.0 function calls. If you want to call some of the more advanced functions from 4.3, use the global pointer to the internally allocated `BHand` object. Here is an example of using old code and calling new version 4.3 commands:

```
#include <BHand.h>
void main()
{
      InitSoftware();
      InitHand();
      _pBh->Baud(19200);
      Close( ‘G’ );
}
```

Initializing the 4.3 Library and using 1.0 function calls is not possible. In general, we recommend modifying existing code to use the 4.3 function format.

# 4 Example Programs

## 4.1 Supervisory Mode Examples

### *4.1.1 Supervisory.cpp*

The following program was generated automatically by the BHControl Interface by clicking the Generate-C++ button. This code is an example of how to use the Supervisory mode to control finger motions. This code closes the Spread and then the fingers in the sequence F1, F3 and then F2. The fingers then open in the opposite sequence F2, F3 and then F1 and terminate control of the Spread motor.

```
////////////////////////////////////////////////////////
//                                                    //
//           Automatically Generated C++ Code         //
//           FROM BHandControl INTERFACE              //
//                                                    //
//                   Supervisory Mode                 //
//                                                    //
////////////////////////////////////////////////////////


#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <conio.h>
#include "BHand.h"

BHand  bh;           // Handles all hand communication
int    value;        // Hand parameter obtained with Get
int    result;       // Return value (error) of all BHand calls


////////////////////////////////////////////////////////
// Error Handler - called whenever result!=0
// Feel free to customize this function

void Error(void)
{
     printf( "ERROR: %d\n%s\n", result, bh.ErrorMessage(result) );
     exit(0);
}


////////////////////////////////////////////////////////
//  Initialize hand, set timeouts and baud rate

void Initialize(void)
{
     if( result=bh.InitSoftware(1,THREAD_PRIORITY_TIME_CRITICAL) )
          Error();

     if( result=bh.ComSetTimeouts(0,100,15000,100,5000) )
          Error();

     if( result=bh.Baud(9600) )
          Error();
```

```
     if( result=bh.InitHand("") )
          Error();
}


////////////////////////////////////////////////////////
//  Execute commands, return 1 if interrupted with a key

int Execute(void)
{
     printf( "Press Any Key to Abort..." );

     if( result=bh.Close( "S" ) )
          Error();
     if( _kbhit() )
          { _getch(); return 1; }

     if( result=bh.Close( "1" ) )
          Error();
     if( _kbhit() )
          { _getch(); return 1; }

     if( result=bh.Close( "3" ) )
          Error();
     if( _kbhit() )
          { _getch(); return 1; }

     if( result=bh.Close( "2" ) )
          Error();
     if( _kbhit() )
          { _getch(); return 1; }

     if( result=bh.Open( "2" ) )
          Error();
     if( _kbhit() )
          { _getch(); return 1; }

     if( result=bh.Open( "3" ) )
          Error();
     if( _kbhit() )
          { _getch(); return 1; }

     if( result=bh.Open( "1" ) )
          Error();
     if( _kbhit() )
          { _getch(); return 1; }

     if( result=bh.StopMotor( "S" ) )
          Error();
     if( _kbhit() )
          { _getch(); return 1; }

     return 0;
}


////////////////////////////////////////////////////////
```

```
//  Main function - initialize, execute

void main(void)
{
      printf( "Initialization..." );
      Initialize();
      printf( " Done\n" );

      printf( "Executing - " );
      Execute();
      printf( " Done\n" );
}
```

### *4.1.2 Example of Instantiating Two (2) BHand Objects: bh1 & bh2*

```
////////////////////////////////////////////////////////
//                                                    //
//       Example of Instantiating 2 BHand Objects     //
//                                                    //
//                                                    //
////////////////////////////////////////////////////////


#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <conio.h>
#include "BHand.h"


// Instantiate two objects of type BHand, one per serial port
BHand  bh1;                 // Handles all hand 1 communication
BHand  bh2;                 // Handles all hand 2 communication

int    result;       // Return value (error) of all BHand calls


////////////////////////////////////////////////////////
// Error Handler for hand 1 - called whenever result!=0
// Feel free to customize this function

void Error1(void)
{
     printf( "ERROR: %d\n%s\n", result, bh1.ErrorMessage(result) );
     exit(0);
}

////////////////////////////////////////////////////////
// Error Handler for hand 2 - called whenever result!=0
// Feel free to customize this function

void Error2(void)
{
     printf( "ERROR: %d\n%s\n", result, bh2.ErrorMessage(result) );
     exit(0);
}

////////////////////////////////////////////////////////
//  Initialize hand 1, set timeouts and baud rate

void Initialize1(void)
{
     if( result=bh1.InitSoftware(1,THREAD_PRIORITY_TIME_CRITICAL) )
          Error1();

     if( result=bh1.ComSetTimeouts(0,100,15000,100,5000) )
          Error1();

     if( result=bh1.Baud(9600) )
          Error1();
```

```
     if( result=bh1.InitHand("2") )
          Error1();
}
////////////////////////////////////////////////////
//  Initialize hand 2, set timeouts and baud rate

void Initialize2(void)
{
     if( result=bh2.InitSoftware(2,THREAD_PRIORITY_TIME_CRITICAL) )
          Error2();

     if( result=bh2.ComSetTimeouts(0,100,15000,100,5000) )
          Error2();

     if( result=bh2.Baud(9600) )
          Error2();

     if( result=bh2.InitHand("2") )
          Error2();
}

////////////////////////////////////////////////////
//  Execute commands, return 1 if interrupted with a key

int Execute(void)
{

     printf( "Press Any Key to Abort..." );


// Execute commands on hand 1
     if( result=bh1.Command ( "2hi" ) )
          Error1();
     if( _kbhit() )
          { _getch(); return 1; }

     if( result=bh1.Command ( "2c" ) )
          Error1();
     if( _kbhit() )
          { _getch(); return 1; }

     if( result=bh1.Command ( "2o" ) )
          Error1();
     if( _kbhit() )
          { _getch(); return 1; }


// Execute commands on hand 2
     if( result=bh2.Command ( "2hi" ) )
          Error2();
     if( _kbhit() )
          { _getch(); return 1; }

     if( result=bh2.Command ( "2c" ) )
          Error2();
     if( _kbhit() )
          { _getch(); return 1; }
```

```
      if( result=bh2.Command ( "2o" ) )
            Error2();
      if( _kbhit() )
            { _getch(); return 1; }

      return 0;
}


////////////////////////////////////////////////////////
//  Main function - initialize, execute

void main(void)
{
      printf( "Initialization..." );
      Initialize1();        // Initialize hand 1
      Initialize2();        // Initialize hand 2
      printf( " Done\n" );

      printf( "Executing - " );
      Execute();            // Execute commands on both hands
      printf( " Done\n" );
}
```

### 4.2 RealTime Mode Example: *RealTime.cpp*

The following program was generated automatically by the BHControl Interface by clicking the Generate-C++ button. This code is an example of how to use the RealTime mode to control finger velocities as a function of time. Each of the fingers moves in a sinusoidal motion with a phase shift from the other fingers. This program saves all of the feedback and control data to a file which may be read.

```
////////////////////////////////////////////////////////
//                                                    //
//           Automatically Generated C++ Code         //
//           FROM BHandControl INTERFACE              //
//                    RealTime Mode                   //
//                                                    //
////////////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <conio.h>
#include "BHand.h"

BHand  bh;           // Handles all hand communication
int    value;        // Hand parameter obtained with Get
int    result;       // Return value (error) of all BHand calls

int    szbuffer;     // Size of data buffers
int*   pdata[4][8];  // Pointers to data buffers


////////////////////////////////////////////////////////
// Error Handler - called whenever result!=0
// Feel free to customize this function

void Error(void)
{
     printf( "ERROR: %d\n%s\n", result, bh.ErrorMessage(result) );
     exit(0);
}


////////////////////////////////////////////////////////
//  Initialize hand, set timeouts and baud rate

void Initialize(void)
{
     if( result=bh.InitSoftware(1,THREAD_PRIORITY_TIME_CRITICAL) )
          Error();

     if( result=bh.ComSetTimeouts(0,100,15000,100,5000) )
          Error();

     if( result=bh.Baud(9600) )
          Error();

     if( result=bh.InitHand("") )
          Error();
}
```

```
////////////////////////////////////////////////////////
//  Execute commands, return 1 if interrupted with a key

int Before(void)
{
     printf( "Press Any Key to Abort..." );

     if( result=bh.Command("sc") )
          Error();
     if( _kbhit() )
          { _getch(); return 1; }

     if( result=bh.Command("go") )
          Error();
     if( _kbhit() )
          { _getch(); return 1; }

     return 0;
}


////////////////////////////////////////////////////////
//  Set parameters, allocate data buffers, load files

void PrepareRealTime(void)
{
     szbuffer = 1000;

     for( int m=0; m<4; m++ )
          for( int v=0; v<8; v++ )
          {
               pdata[m][v] = new int[szbuffer];
               memset((void*)pdata[m][v], 0, szbuffer*sizeof(int));
          }

     if( result=bh.RTSetFlags( "123",1,1,0,0,1,0,1,0,1,0,0))
          Error();

}


////////////////////////////////////////////////////////
//  Run RealRime loop, return 1 if interrupted with a key

int RunRealTime(void)
{
     double var[4][7];
     int N=0;
     DWORD  time, tmstart;
     bool terminate=false;
     int pos, m;

     bh.RTStart( "123" );
     tmstart = GetTickCount();
     bh.RTUpdate();
```

```
	printf( "Press Any Key to Abort..." );
	while( !terminate && !_kbhit() )
	{
		time = GetTickCount() - tmstart;

		pos = N%szbuffer;
		for( m=0; m<4; m++ )
		{
			pdata[m][0][pos] = bh.RTGetPosition( m+'1' );
			pdata[m][1][pos] = bh.RTGetVelocity( m+'1' );
			pdata[m][2][pos] = bh.RTGetStrain( m+'1' );
			pdata[m][3][pos] = bh.RTGetDeltaPos( m+'1' );
			pdata[m][5][pos] = (int)time;
		}
		pos = (N+1)%szbuffer;

		for( int m=0; m<4; m++ )
		{
			var[m][0] = bh.RTGetPosition( m+'1' );
			var[m][1] = bh.RTGetVelocity( m+'1' );
			var[m][2] = bh.RTGetStrain( m+'1' );
			var[m][3] = bh.RTGetDeltaPos( m+'1' );
			var[m][4] = pdata[m][4][N%szbuffer];
			var[m][5] = (double)time;
			var[m][6] = (double)N;
		}

		value = (int)((65.00) * (sin( ((var[0][5]) / (3000.00)) *
			((2.00) * (3.14)) )));
		bh.RTSetVelocity( '1', value );
		pdata[0][6][pos] = value;
		value = (int)((65.00) * (sin( (((var[1][5]) / (3000.00)) *
			((2.00) * (3.14))) - (3.14) )));
		bh.RTSetVelocity( '2', value );
		pdata[1][6][pos] = value;
		value = (int)((65.00) * (sin( (((var[2][5]) / (3000.00)) *
			((2.00) * (3.14))) - ((3.14) / (2.00)) )));
		bh.RTSetVelocity( '3', value );
		pdata[2][6][pos] = value;
		terminate = (0<(int)((var[0][5]) > (10000.00)));

		N++;
		bh.RTUpdate();
	}

	bh.RTAbort();
	if( _kbhit() )
		{ _getch(); return 1; }
	else
		return 0;
}


////////////////////////////////////////////////////
//  Execute commands, return 1 if interrupted with a key

int After(void)
{
```

```
      printf( "Press Any Key to Abort..." );

      if( result=bh.Command("t") )
            Error();
      if( _kbhit() )
            { _getch(); return 1; }

      return 0;
}

////////////////////////////////////////////////////////////
//  Save all buffers into a text file

void SaveData(char* name)
{
      FILE* fp = fopen(name, "wt");
      if( !fp )
            return;

      for( int r=0; r<szbuffer; r++ )
      {
            for( int v=0; v<8; v++ )
            {
                  for( int m=0; m<4; m++ )
                        fprintf(fp, "%d ", pdata[m][v][r]);
                  fprintf(fp, "  ");
            }
            fprintf(fp, "\n");
      }

      fclose(fp);
}


////////////////////////////////////////////////////////////
//  Main function - initialize, execute

void main(void)
{
      printf( "Initialization..." );
      Initialize();
      printf( " Done\n" );

      printf( "Before - " );
      if( Before() )
            return;
      printf( " Done\n" );

      PrepareRealTime();

      printf( "RealTime Loop - " );
      if( RunRealTime() )
            return;
      printf( " Done\n" );

      SaveData( "Sinewave" );
}
```

## Appendix A Bhand.h

```
//////////////////////////////////////////////////////////////////
//
//                          (C) Barrett Technology Inc. 1998-2001
//
//
//    Header file for version 4.XX of the BHand C++ library.
//  "BHand.h" must be included in the user program, and "BHand.lib"
//  must be linked to the project.
//


/*****************************************************************/
//                                Revision History
//v4.01 - ET 990818 added BHMODE_RETURN functionality to Library
//                This allows the user to send commands to the Library
without
//                sending commands to the BarrettHand.
//
//v4.1 - BZ - 991217: Added RTDumpFeedback() function
//
//v4.2 - BZ - July 2001: Added TorqueOpen(), TorqueClose(), PSet(), PGet()
//                RTGetTemp(), and a new RTSetFlags() to set all 11 flags
//
//v4.3 - ET July 2001: Multiple Hands Allowed, one per comport
//                removed _BHInitialized flag
//                added BHand* _BHandArray[BH_MAXPORT] with a pointer to
//                      to the hand on each comport
//                added int SelectHand(comport): used to select the
//                      hand that ver 1.0 functions refer to
//                modified ver 1.0 InitSoftware() to take comport parameter
//                      default = 1
//                added BHERR_PORTOUTOFRANGE error message
//                added comPort field in BHand class
//                changed unsigned char to int in ComInitialize()
/*****************************************************************/


#ifndef _INCLUDE_BHAND_H_JUST_ONCE_
#define _INCLUDE_BHAND_H_JUST_ONCE_

#include <windows.h>


// callback function type
typedef     void (*BHCallback)( class BHand* );


// char buffer size, max comport
#define BH_MAXCHAR                          5000
#define BH_MAXPORT                          256


// Software Library Error Messages (hand errors are positive)
#define BHERR_BHANDEXISTS           -1
#define BHERR_OPENCOMMPORT          -2
#define BHERR_GETCOMMSTATE          -3
#define BHERR_SETCOMMSTATE          -4
#define BHERR_SETCOMMTIMEOUT  -5
#define BHERR_THREADSTART           -6
#define BHERR_THREADPRIORITY  -7
#define BHERR_WRITECOM              -8
#define BHERR_READCOM               -9
```

```
#define BHERR_BADRESPONSE           -10
#define BHERR_CLEARBUFFER           -11
#define BHERR_TIMEOUT               -12
#define BHERR_NOTCOMPLETED          -13
#define BHERR_PENDING               -14
#define BHERR_NOTINITIALIZED  -15
#define BHERR_BADPARAMETER          -16
#define BHERR_LONGSTRING            -17
#define BHERR_OUTOFRANGE            -18
#define BHERR_MOTORINACTIVE         -19
#define BHERR_PORTOUTOFRANGE  -20


// Requests to Communications Thread
#define BHREQ_EXIT                          1
#define BHREQ_REALTIME              2
#define BHREQ_SUPERVISE             3
#define BHREQ_CLEAR                         4


// Synchronization Modes
#define BHMODE_SYNC                         1
#define BHMODE_ASYNCNOW             2
#define BHMODE_ASYNCWAIT            3
#define BHMODE_RETURN               4



//////////////////////////////////////////////////////////////////
// Class BHand declaration

class BHand
{
      friend DWORD _stdcall _ComThreadFunction( void* pHand);

public:
      BHand() {};
      ~BHand();
      const char* ErrorMessage(int err);


      ////////////////////////////////////////////////////////////
      // Serial communications

      int         ComInitialize( int comport, int priority );
      int         ComSetTimeouts( DWORD readInterval, DWORD readMultiplier,
DWORD

readConstant,
                                          DWORD writeMultiplier,

DWORD writeConstant );
      int         ComSetBaudrate( DWORD baud );
      int         ComRequest( int requestNumber );
      bool  ComIsPending( void );
      int         ComWaitForCompletion( DWORD timeout );
      bool  ComClear( void );
      int         ComRead( char* buffer, int size );
      int         ComWrite( char* buffer, int size );

      int         syncMode;
      DWORD requestTimeout;
      BHCallback pCallback;
```

```
      ////////////////////////////////////////////////////////////
      // Interactive commands from version 1.0

      int         Open( char* motor );
      int         Close( char* motor );
      int         StepOpen( char* motor, int value );
      int         StepClose( char* motor, int value );
      int         GoToPosition( char* motor, int value );
      int         GoToDifferentPositions( int value1, int value2, int
value3, int value4 );
      int         GoToHome( void );
      int         StopMotor( char* motor );
      int         Set( char* motor, char* parameter, int value );
      int         Get( char* motor, char* parameter, int* result );
      int         Save( char* motor );
      int         Load( char* motor );
      int         Default( char* motor );
      int         Temperature( int* result);
      int         InitSoftware( int port, int priority =

THREAD_PRIORITY_TIME_CRITICAL );
      int         InitHand( char* motor );


      ////////////////////////////////////////////////////////////
      // New interactive commands

      int         Delay( DWORD msec );
      int         GoToDefault( char* motor );
      int         Reset( void );
      int         Baud( DWORD baud );
      int         Command( char* send, char* receive=NULL );
      const char* Response( void );
      const char* Buffer( void );
      int         TorqueOpen( char* motor );
      int         TorqueClose( char* motor );
      int         PSet( char* parameter, int value );
      int         PGet( char* parameter, int* result);


      ////////////////////////////////////////////////////////////
      // Real-time (loop) mode commands

      int         RTStart( char* motor );
      int         RTUpdate( bool control=true, bool feedback=true );
      int         RTSetFlags( char* motor, bool LCV, int LCVC, bool LCPG,
                              bool LFV, int LFVC, bool LFS, bool LFAP,

bool LFDP, int LFDPC );
      int         RTSetFlags( char* motor, bool LCV, int LCVC, bool LCPG,
                              bool LFV, int LFVC, bool LFS, bool LFAP,
bool

LFDP, int LFDPC,
                              bool LFT, bool LFDPD);
      int         RTAbort( void );
      char  RTGetVelocity( char motor );
      unsigned char RTGetStrain( char motor );
      int         RTGetPosition( char motor );
      char  RTGetDeltaPos( char motor );
      int         RTGetTemp( void );
      int         RTSetVelocity( char motor, int velocity );
      int         RTSetGain( char motor, int gain );
```

```
      void  RTDumpFeedback(void);

      int         rtFlags[4][7];
      int         rtGlobalFlags[1];
      int         nSend;
      int         nReceive;
      char  rtIn[22];


      ////////////////////////////////////////////////////////////
      // Internal variables - not accessible by the user program

      HANDLE      com;
private:
      HANDLE      thread;
      DWORD threadId;
      int         request;
      HANDLE      requestPending;
      HANDLE      requestComplete;
      DWORD requestBaud;
      char  inbuf[BH_MAXCHAR];
      int         nin;
      char  outbuf[BH_MAXCHAR];
      int         nout;
      int         comErr;
      char  rtOut[8];
      int         rtControl[4][2];
      int         rtFeedback[4][4];
      int         rtGlobalFeedback[1];
      int         comPort;
};


// pointer to hand corresponding to each port-1 (1 : BH_MAXPORT)
extern BHand* _BHandArray[];


// Obsolete fuctions provided for version 1.0 compatibility
int   Open( char motor );
int   Close( char motor );
int   StepOpen( char motor, int value );
int   StepClose( char motor, int value );
int   GoToPosition( char motor, int value );
int   GoToDifferentPositions( int value1, int value2, int value3, int value4
);
int   GoToHome( void );
int   StopMotor( char motor );
int   Set( char motor, char parameter, int value );
int   Get( char motor, char parameter, int* result );
int   Save( char motor );
int   Load( char motor );
int   Default( char motor );
int   Temperature( int* result );
int   InitSoftware( int comport = 1 );
int   InitHand( void );
int   SelectHand( int comport );


// pointer to selected hand for vers 1.0 compatibility
extern BHand* _pBH;


#endif
```

# Appendix B MS-Windows Description of COM Timeout Parameters

## COMMTIMEOUTS

The **COMMTIMEOUTS** structure is used in the **SetCommTimeouts** and **GetCommTimeouts** functions to set and query the time-out parameters for a communications device. The parameters determine the behavior of **ReadFile**, **WriteFile**, **ReadFileEx**, and **WriteFileEx** operations on the device.

```
typedef struct _COMMTIMEOUTS {
    DWORD ReadIntervalTimeout;
    DWORD ReadTotalTimeoutMultiplier;
    DWORD ReadTotalTimeoutConstant;
    DWORD WriteTotalTimeoutMultiplier;
    DWORD WriteTotalTimeoutConstant;
} COMMTIMEOUTS,*LPCOMMTIMEOUTS;
```

**Members**

**ReadIntervalTimeout**

Specifies the maximum time, in milliseconds, allowed to elapse between the arrival of two characters on the communications line. During a **ReadFile** operation, the time period begins when the first character is received. If the interval between the arrival of any two characters exceeds this amount, the **ReadFile** operation is completed and any buffered data is returned. A value of zero indicates that interval time-outs are not used.

A value of MAXDWORD, combined with zero values for both the **ReadTotalTimeoutConstant** and **ReadTotalTimeoutMultiplier** members, specifies that the read operation is to return immediately with the characters that have already been received, even if no characters have been received.

**ReadTotalTimeoutMultiplier**

Specifies the multiplier, in milliseconds, used to calculate the total time-out period for read operations. For each read operation, this value is multiplied by the requested number of bytes to be read.

**ReadTotalTimeoutConstant**

Specifies the constant, in milliseconds, used to calculate the total time-out period for read operations. For each read operation, this value is added to the product of the **ReadTotalTimeoutMultiplier** member and the requested number of bytes.

A value of zero for both the **ReadTotalTimeoutMultiplier** and **ReadTotalTimeoutConstant** members indicates that total time-outs are not used for read operations.

**WriteTotalTimeoutMultiplier**

Specifies the multiplier, in milliseconds, used to calculate the total time-out period for write operations. For each write operation, this value is multiplied by the number of bytes to be written.

**WriteTotalTimeoutConstant**

Specifies the constant, in milliseconds, used to calculate the total time-out period for write operations. For each write operation, this value is added to the product of the **WriteTotalTimeoutMultiplier** member and the number of bytes to be written.

A value of zero for both the **WriteTotalTimeoutMultiplier** and **WriteTotalTimeoutConstant** members indicates that total time-outs are not used for write operations.

**Remarks**

If an application sets **ReadIntervalTimeout** and **ReadTotalTimeoutMultiplier** to MAXDWORD and sets **ReadTotalTimeoutConstant** to a value greater than zero and less than MAXDWORD, one of the following occurs when the **ReadFile** function is called:

- If there are any characters in the input buffer, **ReadFile** returns immediately with the characters in the buffer.
- If there are no characters in the input buffer, **ReadFile** waits until a character arrives and then returns immediately.
- If no character arrives within the time specified by **ReadTotalTimeoutConstant**, **ReadFile** times out.

**QuickInfo**

**Windows NT:** Requires version 3.1 or later.
**Windows:** Requires Windows 95 or later.
**Windows CE:** Requires version 1.0 or later.
**Header:** Declared in winbase.h.

# INDEX