Ghost API Programming Reference

Rev. 1.1 (Draft)

_____

LEGO Technology Center, LEGO System A/S.

Table of Contents

1 Overview

Welcome to the Ghost C API! This portable API allows applications to easily instantiate and use a Ghost communication stack.

The Ghost API comes as a DLL (Win32) or a shared library (MacOS), and it depends on the Ghost core libraries.

To use the Ghost API from your C/C++ application, you will just have to include the GhostAPI.h header file, and add the GhostAPI import/stub library to the project. The GhostAPI DLL plus the Ghost core libraries will have to be present in the same directory where your application executable is.

When using the Ghost API, you'll have to follow these steps:

1. Instantiate a Ghost stack

2. Select a communication device (e.g., a tower) among the available ones

3. Open the stack for communication

4. Select a notification mode (Wait or NoWait)

5. Create and submit command queues

6. Close and destroy the Stack

1.1 Ghost Communication Stack

A Ghost stack is a set of three software layers, which take care of communication between a computer and an external LEGO device. Each layer of the stack is bound to the underneath layer. The three layers are called **Port**, **Prototol** and **Session**.

It's important to understand that the Ghost communication stack only takes care of data transport. It doesn't have any knowledge of specific PBrick commands (since there are several commands sets for the various types of PBricks LEGO produces), but if you submit to Ghost a command in form of a bytestream, Ghost will send the command for you, retrieve the answer and notify you of the result.

If you want to compile a PBrick program, or otherwise translate a PBrick command into a bytestream, you should look into the LASM library.

1.1.1 Port Layer

This layer abstracts the communication device type and the OS. Right now, serial port implementations are available for the Win32, PSX and MacOS platform. USB tower port layers have been implemented for Win32 and for the MacOS.

Protocol Layer

Session Layer

1.1.2 Protocol Layer

This layer abstracts the communication protocol used in the stack. It takes care of packet forming and transmission, acknowledge receipt and retransmissions. Currently there are protocol implementations for the IR (RCX) protocol and for the Radio (Cybermaster) protocol. In order to be able to use the protocol layer, is has to be bound to a Port layer.

Port Layer

Session Layer

1.1.3 Session Layer

The session layer allows you to submit queues of commands to the communication stack. Commands can be submitted to two queues: the execute queue and the download queue. The session layer takes care of interleaving commands from the two queue according to the current interleave settings.

Port Layer

Protocol Layer

1.2 Instantiating a Ghost Stack

When programming with the Ghost API, the first thing you'll want to do is to instantiate a Ghost stack. To do so, call GhCreateStack.

When you are done using a hstack, make sure you destroy it by calling GhDestroyStack.

1.3 Selecting a notification mode

The Ghost stack can work in two modes, **Wait** and **NoWait**. In Wait mode, command queue submissions will only return when all the commands in the queue have been processed (or an error occurs). In NoWait mode, both GhExecute and GhDownload will return immediately with a return code of PBKOK_PENDING. You'll also have to provide a notification function which the GhostAPI will call at completion of every command in the queue, **and** when the queue itself is completely processed.

To set notification mode to Wait, call GhSetWaitMode. To set notification mode to NoWait, call GhSetNoWaitMode.

Note As you might have noticed, you can pass also a notification function to GhSetWaitMode. This allow you to be notified of command completion even in wait mode. No matter whether you are in Wait or NoWait mode, your notification function will run on a different thread than the thread that submitted the command queue. This will probably require your application to implement some mechanism to synchronize access to common data from multiple threads.

See Also
Notification Handling

1.4 Notification Handling

1.5 Error Handling

All functions in the Ghost API return an error code of type PBKRESULT. In case of success, the error code will be PBKOK or (for NoWait operations) PBKOK_PENDING. In case of failure, the returned code will be a constant of type PBK_ERR_...

There are a couple of handy inline functions to test a function's return code: PBK_SUCCEEDED and PBK_FAILED. So your code might look like the example below.

All the error constants and utility functions are defined in pbkerror.h, which is included by ghostapi.h

1.6 Wait vs NoWait Mode

Once you have created and initialized a communication stack object, you can set its communication mode to either Wait or NoWait. You do this by calling the GhSetWaitMode or the GhSetNoWaitMode functions. Default mode is Wait.

So, what do these modes mean?

In Wait mode, both GhExecute and GhDownload calls are blocking, i.e., they return to you only after all the commands you submitted have been processed or an error occurred. To figure out what commands have been executed, you can walk the command queue and get info about each command (GhGetFirstCommand, GhGetNextCommand, GhGetCommandInfo).

If you set Ghost in NoWait mode, GhExecute/GhDownload will return you immediately with a code of PBKOK_PENDING, and the commands will be submitted in the background. In order to know what happened to them, you have to pass to GhSetNoWaitMode the address of a notification function that Ghost will call for each command that gets executed, and when the whole request queue has been completely processed (or an error/abort occurred).

To find out about the context in which the notification function is called, see Differences between MacOS and Windows

See Also
Differences between MacOS and Windows

1.7 Differences between MacOS and Windows

The Ghost API exposes to the application the very same programming interface in the Windows and MacOS version. The main difference is in how asynchronous I/O is treated in the two platforms. While Ghost for Win32 makes full use of preemptive multitasking, Ghost for MacOS uses cooperative threads instead.

For your application, the above difference means that:

- On Win32, your notification functions will be called on threads other than your application's thread(s).
- On the MacOS, make sure that you relinquish control to other threads by calling the MacOS API function YieldToAnyThread in your app's event loop. If you are writing a Sioux-enabled console application, you might have to manually fiddle with Sioux's event loop.

1.8 Opening the stack for communication

Once you have a valid stack, and have selected a device you want to use for communication, you'll have to actually open the stack. You do so by calling GhOpen. This function binds the stack to the currently selected device (i.e., the device selected by the last successful call to GhSelectFirstDevice or GhSelectNextDevice). Remember to close a stack after usage by calling GhClose.

1.9 Selecting a Communication Device

To select a communication device, you have to instantiate a Ghost stack first. When you have a valid stack handle, you can enumerate and select the available devices by calling GhSelectFirstDevice and GhSelectNextDevice. If you have only a tower attached to the computer (the most likely case), only GhSelectFirstDevice will return success; further calls to GhSelectNextDevice will return PBK_ERR_NOMORE.

Note that the above functions will enumerate only the devices of the type supported by the port layer instantiated by the stack: if you create an hstack specifying that you want a

Serial port layer ("LEGO.Pbk.CommStack.Port.RS232"), well that hstack will allow you to enumerate only serial towers. To enumerate USB towers you'll have to create a hstack supporting a USB tower port ("LEGO.Pbk.CommStack.Port.USB").

2 Reference

2.1 Function Reference

2.1.1 Ghost Stack Handling Functions

2.1.1.1 GhClose

Close the currently selected device

Declaration

```
PBKRESULT   PBK_EXPORT GhClose( GHSTACK hstack )
```

Parameters

hstack (IN) Handle of the stack to be closed.

Return Value

PBKOK or an error code (PBK_ERR_CLOSED if the stack was not open).

See Also

GhOpen

2.1.1.2 GhCreateStack

Creates a Ghost communication stack

Declaration

```
PBKRESULT   PBK_EXPORT GhCreateStack(   const char      *pszport,
                                        const char      *pszprotocol,
                                        const char      *pszsession,
                                        GHSTACK         *phstack
)
```

Parameters

pszport (IN) port implementation required

pszprotocol (IN) protocol implementation required

pszsession (IN) session implementation required

phstack (OUT) Ptr to the stack handle created by this function

Return Value

PBKOK or an error code

Remarks

Supported implementations so far:

Serial port: “LEGO.Pbk.CommStack.Port.RS232” USB port: “LEGO.Pbk.CommStack.Port.USB”

IR (RCX) Protocol: “LEGO.Pbk.CommStack.Protocol.IR” Radio (Cybermaster) Protocol: “LEGO.Pbk.Protocol.Radio”

Session (generic): “LEGO.Pbk.CommStack.Session”

Example
The following example creates a communication stack supporting the USB tower and the IR (RCX-Scout) protocol.

```
PBKRESULT pbkrc;
GHSTACK   hstack;

pbkrc = GhCreateStack( "LEGO.Pbk.CommStack.Port.USB",
                       "LEGO.Pbk.CommStack.Protocol.IR",
                       "LEGO.Pbk.CommStack.Session",
                       &hstack );
```

See Also
GhDestroyStack

2.1.1.3 GhDestroyStack

Destroys a communication stack freeing all its associated memory

Declaration

```
PBKRESULT   PBK_EXPORT GhDestroyStack(  GHSTACK  hstack )
```

Parameters
hstack a valid stack handle created by GhCreateStack

Return Value
PBKOK or an error code

See Also
GhCreateStack

2.1.1.4 GhGetPortType

Return the port type of the given stack

Declaration

```
PBKRESULT   PBK_EXPORT GhGetPortType(   GHSTACK     hstack,
                                         char       *psztype,
                                         uint32      ulbufsize      )
```

Parameters
hstack (IN) Stack handle

psztype OUT

ulbufsize (IN) Size of the buffer pointed by psztype.

Return Value
PBKOK or error code (PBK_ERR_BADPARM, PBK_ERR_OVERFLOW)

Remarks
This is the same string as passed to GhCreateStack

See Also

GhCreateStack, GhGetProtocolType, GhGetSessionType

2.1.1.5 GhGetProtocolType

Return the protocol type of the given stack

Declaration

```
PBKRESULT   PBK_EXPORT GhGetProtocolType(   GHSTACK     hstack,
                                            char        *psztype,
                                            uint32       ulbufsize
)
```

Parameters

hstack (IN) Stack handle

psztype OUT

ulbufsize (IN) Size of the buffer pointed by psztype.

Return Value

PBKOK or error code (PBK_ERR_BADPARM, PBK_ERR_OVERFLOW)

Remarks

This is the same string as passed to GhCreateStack

See Also

GhCreateStack, GhGetPortType, GhGetSessionType

2.1.1.6 GhGetSessionType

Return the session type of the given stack

Declaration

```
PBKRESULT   PBK_EXPORT GhGetSessionType(    GHSTACK     hstack,
                                            char        *psztype,
                                            uint32       ulbufsize
)
```

Parameters

hstack (IN) Stack handle

psztype OUT

ulbufsize (IN) Size of the buffer pointed by psztype.

Return Value

PBKOK or error code (PBK_ERR_BADPARM, PBK_ERR_OVERFLOW)

Remarks

This is the same string as passed to GhCreateStack

See Also

GhCreateStack, GhGetPortType, GhGetProtocolType

2.1.1.7 GhOpen

Opens the currently selected device

Declaration

```
PBKRESULT   PBK_EXPORT GhOpen(  GHSTACK hstack )
```

Parameters

hstack (IN) Handle of the stack to be opened.

Return Value

PBKOK or an error code (PBK_ERR_INVALIDSTATE if no device has been previously selected)

Remarks

Before calling this function the caller needs to select a device via GhSelectFirst and GhSelectNext.

It's a good thing to close the session when done with it. You may do that by calling GhClose.

GhClose, GhSelectFirstDevice, GhSelectNextDevice.

2.1.2 Device Enumeration Functions

2.1.2.1 GhDiagSelectFirstPort

Find and select the first port.

Declaration

```
PBKRESULT   PBK_EXPORT GhDiagSelectFirstPort(   GHSTACK      hstack,
                                                char
*pszportname,
                                                uint32
ulbufsize      )
```

Parameters

hstack (IN) Stack handle

pszportname (OUT) Ptr to buffer filled with the name of the selected port device

ulbufsize (IN) Size of the buffer pointed by pszdevicename.

Return Value

PBKOK or error code

Remarks

This function does port enumeration at port layer, not through the session/protocol layers. This means that this function (and GhDiagSelectNextPort) just forces selection of the existing ports without trying to figure out whether the port is in use, or (for serial ports) whether a tower is attached to the given port. Therefore, a subsequent GhOpen may very well fail.

Remember that the recommended way to enumerate and select available device is by calling GhSelectFirst.

See Also

GhCreateStack, GhDiagSelectNextPort, GhSelectFirstDevice, Selecting a Communication Device

2.1.2.2 GhDiagSelectNextPort

Find and select the next available port.

Declaration

```
PBKRESULT   PBK_EXPORT GhDiagSelectNextPort(    GHSTACK      hstack,
                                                char
*pszportname,
                                                uint32
ulbufsize      )
```

Parameters

hstack	(IN)	Stack handle
ulbufsize	(IN)	Size of the buffer pointed by pszdevicename.

Return Value
PBKOK or error code

Remarks
Call this function while enumerating ports, after GhDiagSelectFirstPort. Use this function for diagnostic purposes only - normally you should use GhSelectFirstDevice/GhSelectNextDevice instead.

See Also
GhCreateStack, GhSelectFirstDevice

2.1.2.3 GhDiagSelectSpecificPort

Forces Ghost to select a specific port given the port name

Declaration

```
PBKRESULT   PBK_EXPORT GhDiagSelectSpecificPort(    GHSTACK
hstack,
                                                    char
*pszportname     )
```

Parameters

hstack	(IN)	Stack handle
pszportname	(IN)	Ptr to buffer containing the port name to be selected

Return Value
PBKOK or error code

Remarks
The proper way to enumerate and select devices in the Ghost API is by using the GhSelectFirstDevice/GhSelectNextDevice pair. As the difference in name indicates, this function allow you to force Ghost using a specific “port name” but you have no guarantee that a valid device is attached to the given port.

If you decide to use this function, don't insert it within a GhSelectFirstDevice /GhSelectNextDevice cycle, otherwise the results may be unpredictable (either some devices will be counted twice or some will be skipped).

See Also
GhCreateStack, GhSelectFirstDevice, GhSelectNextDevice, Selecting a Communication Device

2.1.2.4 GhGetDeviceName

Return the name of the currently selected device

Declaration

```
PBKRESULT   PBK_EXPORT GhGetDeviceName(     GHSTACK     hstack,
                                            char
*pszdevicename,
                                            uint32       ulbufsize
)
```

Parameters

hstack	(IN) Stack handle
pszdevicename	OUT
ulbufsize	(IN) Size of the buffer pointed by pszdevicename.

Return Value
PBKOK or error code (PBK_ERR_BADPARM, PBK_ERR_OVERFLOW)

Remarks
To select a device you can use GhSelectFirstDevice and GhSelectNextDevice

GhCreateStack, GhSelectFirstDevice, Selecting a Communication Device

2.1.2.5 GhSelectFirstDevice

Find and select the first available device

Declaration

```
PBKRESULT   PBK_EXPORT GhSelectFirstDevice( GHSTACK      hstack,
                                            char
*pszdevicename,
                                            uint32       ulbufsize
)
```

Parameters

hstack	(IN) Stack handle
pszdevicename	(OUT) Ptr to buffer filled with the name of the selected port device
ulbufsize	(IN) Size of the buffer pointed by pszdevicename.

Return Value
PBKOK or error code

See Also
GhCreateStack, GhSelectNextDevice, Selecting a Communication Device

2.1.2.6 GhSelectNextDevice

Find and select the next available device

Declaration

```
PBKRESULT   PBK_EXPORT GhSelectNextDevice(  GHSTACK      hstack,
                                            char
*pszdevicename,
                                            uint32       ulbufsize
)
```

Parameters

hstack	(IN) Stack handle
pszdevicename	OUT
ulbufsize	(IN) Size of the buffer pointed by pszdevicename.

Return Value
PBKOK or error code

Remarks
GhCreateStack, GhSelectFirstDevice

2.1.3 Queue and Command Handling Functions

2.1.3.1 GhAppendCommand

Appends a command to the given command queue.

Declaration

```
PBKRESULT   PBK_EXPORT GhAppendCommand( GHQUEUE      hqueue,
                                        uchar       *pcommanddata,
                                        uint32       ulcommandlen,
                                        uint32
ulexpectedreplylen     )
```

Parameters

hqueue	(IN) Queue handle (created via GhCreateCommandQueue)
pcommanddata	(IN) Ptr to command buffer (command + parameters)
ulcommandlen	(IN) Length of the command buffer
ulexpectedreplylen	(IN) Length of the expected reply to this command

Return Value
PBKOK or error code (PBK_ERR_NOMEMORY, PBK_ERR_BADPARM, PBK_ERR_INUSE)

Remarks
The command buffer (and len) doesn't include any protocol-specific data (like e.g. preamble, byphase, checksum, etc). Those are automatically added by the protocol layer. Similarly, the expected reply len should not take into account the protocol overhead but just the actual data.

This function allocates a new buffer for the command data and copies the data pointed by pcommanddata. So you don't have to worry about keeping the pcommanddata buffer around.

If the queue passed to this function is currently in use by a communication stack (i.e., if it has been passed to either GhExecute or GhDownload, and it's not completed yet), this function returns PBK_ERR_INUSE.

2.1.3.2 GhCheckQueueInUse

See if the given queue is currently in use by this or some other communication stack.

Declaration

```
PBKRESULT PBK_EXPORT GhCheckQueueInUse( GHQUEUE hqueue, uint32
*pulinuse )
```

Parameters

hqueue (IN) A valid queue handle

pulinuse (OUT) Ptr to returned flag.

Return Value

PBKOK or error code. If the function is successful, the value pointed by pulinuse is set to: nonzero if the queue is currently in use, or to zero if the queue is not currently used.

Remarks

A queue that is currently in use cannot be: (a) submitted again to GhDownload or GhExecute, nor (b) be destroyed by GhDestroyQueue. Also, you cannot append commands to a queue that is currently in use.

See Also

GhCreateCommandQueue, GhExecute, GhDownload

2.1.3.3 GhCreateCommandQueue

Create a command queue (containing one command to start with) and return handle

Declaration

```
PBKRESULT   PBK_EXPORT GhCreateCommandQueue( GHQUEUE *phqueue   )
```

Parameters

phqueue (OUT) Ptr to the queue handle. Initialized by this function

Return Value

PBKOK or an error code

Remarks

In order to send commands through the Ghost communication stack, you have to:

1. Create a command queue via GhCreateCommandQueue. 2. Append one or more commands to the queue (via GhAppendCommand). 3. Call GhExecute or GhDownload to submit your command queue. 4. When you are done, you should destroy the command queue by calling GhDestroyCommandQueue.

See Also

GhDestroyCommandQueue, GhAppendCommand, GhExecute, GhDownload

2.1.3.4 GhDestroyCommandQueue

Release a command queue

Declaration

```
PBKRESULT   PBK_EXPORT GhDestroyCommandQueue( GHQUEUE hqueue )
```

Parameters

hqueue (IN) Queue handle.

Return Value

PBKOK or an error code (PBK_ERR_BADPARM, PBK_ERR_INUSE)

Remarks

This function frees up a command queue and all the associated commands. Therefore:

Do **not** call this function if the command queue is currently in use by the communication stack - if you do, the function will return PBK_ERR_INUSE.

Do **not** use hqueue in any way after this function returns.

See Also

GhCreateCommandQueue

2.1.3.5 GhDownload

Submit a command queue on the DOWNLOAD queue

Declaration

```
PBKRESULT   PBK_EXPORT GhDownload( GHSTACK hstack, GHQUEUE hqueue   )
```

Parameters

hstack (IN) A valid communication stack handle

hqueue (IN) A valid command queue handle. It must refer to a non-empty queue.

Return Value

If successful, this function returns PBKOK (Wait mode) or PBKOK_PENDING (no wait mode). If an error occurs the relative error code is returned.

Remarks

First you create a command queue by calling GhCreateCommandQueue. Then, you append commands to the queue (GhAppendCommand). When the queue is formed you can call GhDownload or GhExecute to submit the commands to the communication stack.

If the current notification mode is WAIT, this function won't return until all the commands in the queue have been processed, or an error occurs.

If the current notification mode is NO WAIT, this function will return immediately a PBKOK_PENDING code (unless some error prevents the queue to be submitted to the communication stack).

In both WAIT and NOWAIT cases, if GhExecute is successful and you have specified a notification function (see GhSetWaitMode and GhSetNoWaitMode), Ghost will then notify your application when individual commands in the queue are processed, and when the whole queue request is completed (or aborted for any reason).

[Win32-specific] Notifications typically happen in a different thread than the thread that submits the requests, so your application has to be thread-safe.

You cannot submit a queue that is already being transmitted by this (or another) stack.

You cannot submit a queue to GhDownload if this stack is already processing a download queue (if you try this you'll get back a PBK_ERR_BUSY).

Before you submit a queue to GhExecute or GhDownload, you can associate a context value to the queue by calling the GhSetQueueContext function. Ghost will pass this value back to your notification function for each notification relative to that queue.

See Also
GhExecute, GhSetWaitMode, GhSetNoWaitMode, GhSetQueueContext, Queue and Command Handling Functions, Notification Handling

2.1.3.6 GhExecute

Submit a command queue on the EXECUTE queue

Declaration

```
PBKRESULT   PBK_EXPORT GhExecute( GHSTACK hstack, GHQUEUE hqueue    )
```

Parameters

hstack	(IN) A valid communication stack handle
hqueue	(IN) A valid command queue handle. It must refer to a non-empty queue.

Return Value
If successful, this function returns PBKOK (Wait mode) or PBKOK_PENDING (no wait mode). If an error occurs the relative error code is returned.

Remarks
First you create a command queue by calling GhCreateCommandQueue. Then, you append commands to the queue (GhAppendCommand). When the queue is formed you can call GhDownload or GhExecute to submit the commands to the communication stack.

If the current notification mode is WAIT, this function won't return until all the commands in the queue have been processed, or an error occurs.

If the current notification mode is NO WAIT, this function will return immediately a PBKOK_PENDING code (unless some error prevents the queue to be submitted to the communication stack).

In both WAIT and NOWAIT cases, if GhExecute is successful and you have specified a notification function (see GhSetWaitMode and GhSetNoWaitMode), Ghost will then notify your application when individual commands in the queue are processed, and when the whole queue request is completed (or aborted for any reason).

[Win32-specific] Notifications typically happen in a different thread than the thread that submits the requests, so your application has to be thread-safe.

You cannot submit a queue that is already being transmitted by this (or another) stack.

You cannot submit a queue to GhExecute if this stack is already executing a queue (if you try this you'll get back a PBK_ERR_BUSY).

Before you submit a queue to GhExecute or GhDownload, you can associate a context value to the queue by calling the GhSetQueueContext function. Ghost will pass this value back to your notification function for each notification relative to that queue.

See Also
GhDownload, GhSetWaitMode, GhSetNoWaitMode, GhSetQueueContext, Queue and Command Handling Functions, Notification Handling

2.1.3.7 GhGetCommandData

Get command data

Declaration

```
PBKRESULT   PBK_EXPORT GhGetCommandData(    GHCOMMAND   hcommand,
                                            uchar
*pcommanddata,
                                            uint32      ulbufsize
)
```

Parameters

hcommand	(IN) Handle of a valid command (retrieved by GhGetFIrstCommand or GhGetNextCommand)
pcommanddata	OUT
ulbufsize	(IN) Size of the buffer pointed by pcommanddata

Return Value
PBKOK or error code (PBK_ERR_BADPARM, PBK_ERR_OVERFLOW)

Remarks
You can call GhGetCommandDataLen to find out the necessary size of the buffer pointed by pcommanddata.

See Also
GhGetFirstCommand, GhGetNextCommand, GhGetCommandDataLen, GhGetCommandReply, GhGetCommandReplyLen

2.1.3.8 GhGetCommandDataLen

Get command data length

Declaration

```
PBKRESULT   PBK_EXPORT GhGetCommandDataLen(     GHCOMMAND   hcommand,
                                                uint32
*pulcommandlen  )
```

Parameters

hcommand	(IN) Handle of a valid command (retrieved by GhGetFIrstCommand or GhGetNextCommand)
pulcommandlen	(OUT) Ptr to `uint32` filled in with the command data len

Return Value
PBKOK or error code (PBK_ERR_BADPARM)

See Also
GhGetFirstCommand, GhGetNextCommand, GhGetCommandData, GhGetCommandReply, GhGetCommandReplyLen

2.1.3.9 GhGetCommandReply

Get command reply

Declaration

```
PBKRESULT   PBK_EXPORT GhGetCommandReply(    GHCOMMAND  hcommand,
                                             uchar      *preplydata,
                                             uint32     ulbufsize
)
```

Parameters

hcommand (IN) Handle of a valid command (retrieved by GhGetFIrstCommand or GhGetNextCommand)

preplydata OUT

ulbufsize (IN) Size of the buffer pointed by pcommanddata

Return Value
PBKOK or error code (PBK_ERR_BADPARM, PBK_ERR_OVERFLOW)

Remarks
You call this function after a command has been completed, to retrieve the reply. The reply size can be retrieved by calling GhGetCommandReplyLen.

If the command has not been completed yet, the function returns PBK_ERR_INVALIDSTATE.

It's better not to use this function to find out whether a command has been completed or not. You should instead rely on the notification function (in NOWAIT mode) or on the return code from GhDownload and GhExecute (in WAIT mode).

If the function succeeds, the buffer pointed by preplydata will contain a copy of the command reply. To find out the reply length (i.e., the number of bytes copied by this function), call GhGetCommandReplyLen.

See Also
GhGetFirstCommand, GhGetNextCommand, GhGetCommandDataLen, GhGetCommandReplyLen, GhDownload, GhExecute

2.1.3.10 GhGetCommandReplyLen

Get command reply length

Declaration

```
PBKRESULT   PBK_EXPORT GhGetCommandReplyLen(    GHCOMMAND   hcommand,
                                                uint32
*pulreplylen    )
```

Parameters

hcommand	(IN) Handle of a valid command (retrieved by GhGetFIrstCommand or GhGetNextCommand)
pulreplylen	(OUT) Ptr to `uint32` filled in with the command reply len

Return Value
PBKOK or error code (PBK_ERR_BADPARM)

Remarks
This function returns the actual reply len (not the expected one). This means that, if the command is not completed, the reply len returned will be 0.

See Also
GhGetFirstCommand, GhGetNextCommand, GhGetCommandData, GhGetCommandDataLen, GhGetCommandReply

2.1.3.11 GhGetFirstCommand

Get the first command in the queue

Declaration

```
PBKRESULT   PBK_EXPORT GhGetFirstCommand( GHQUEUE       hqueue,
                                          GHCOMMAND *phcommand  )
```

Parameters

hqueue	(IN) Queue handle (created via GhCreateCommandQueue)
phcommand	(OUT) Ptr to the returned command handle

Return Value
PBKOK or an error code. If the queue contains no commands, the function returns PBK_ERR_NOMORE

Remarks
This function returns a “handle” to the first command present in the given command queue. You can then pass that handle to a number of functions allowing you to query for various command information (like the command data, the reply, etc).

The command so returned is **not** removed from the queue (you can't remove commands from a command queue).

To walk the command queue, start with GhGetFirstCommand and continue by calling GhGetNextCommand.

You don't have to “free” or release the returned command handle, but DON'T release the command queue while you are still using a command handle belonging to that queue (releasing the queue does release all the commands chained to it).

See Also
GhGetNextCommand, GhGetCommandData, GhGetCommandDataLen, GhGetCommandReply, GhGetCommandReplyLen

2.1.3.12 GhGetNextCommand

Get the next command in the queue

Declaration

```
PBKRESULT   PBK_EXPORT GhGetNextCommand(    GHCOMMAND   hcommand,
                                            GHCOMMAND   *phnext     )
```

Parameters

hcommand (IN) Handle of the previous command.

phnext (OUT) Ptr to the returned (next) command handle

Return Value

PBKOK or an error code. If the queue contains no more commands, the function returns PBK_ERR_NOMORE

Remarks

The command so returned is **not** removed from the queue (you can't remove commands from a command queue).

phnext can point to hcommand - i.e., you can write womething like this:

```
pbkrc   GhGetNextCommand( hcommand, &hcommand );
```

See Also

GhGetFirstCommand, GhGetCommandData, GhGetCommandDataLen, GhGetCommandReply, GhGetCommandReplyLen

2.1.3.13 GhGetQueueContext

Declaration

```
PBKRESULT PBK_EXPORT GhGetQueueContext( GHQUEUE hqueue, void
**pcontext )
```

Parameters

hqueue (IN) a valid queue handle

pcontext (OUT) address of the context value returned

Return Value

PBKOK or an error code (PBK_ERR_BADPARM)

Remarks

Client applications should never need to call this function. The context value is passed back to the notification function as a separate parameter.

See Also

GhSetQueueContext

2.1.3.14 GhSetQueueContext

Declaration

```
PBKRESULT PBK_EXPORT GhSetQueueContext( GHQUEUE hqueue, void
*pcontext )
```

Parameters

hqueue (IN) a valid queue handle

pcontext (IN) anything

Return Value
PBKOK or an error code

Remarks
Once you submit the queue to Ghost and Ghost notifies you of the various commands results, your notification function gets back the context value associated with the queue. Typically you use this function to associate a command queue with a class object or some data structure used by your application.

See Also
GhGetQueueContext

2.1.4 Settings Functions

2.1.4.1 GhDeviceControl

Send a device-specific command to the device attached to the communication stack

Declaration

```
PBKRESULT   PBK_EXPORT GhDeviceControl( GHSTACK      hstack,
                                        char        *pszcommand,
                                        char        *pszresult,
                                        uint32       ulresultsize
                                       )
```

Parameters

hstack (IN) Stack handle

pszcommand (IN) Ptr to command string

pszresult (OUT) Ptr to char buffer filled in with the command reply

ulresultsize (IN) Size of the char buffer pointed by pszresult

Return Value
PBKOK or an error code

Remarks
pszresult can be NULL if the command doesn't have a reply (or if the caller is not interested in the reply).

The commands and their parameters depend on the specific port layer implementation (i.e., the USB tower takes different commands than the serial tower).

Serial tower commands

The serial implementation of the port layer accepts the following commands as parameters for devicecontrol:

- a settings strings compatible with the DOS mode command (eg “baud=9600 parity=N data=8 stop=1”)

or

- dtr=n to either set or reset the dtr signal (n can be 0 or 1) or

o rts=n to either set or reset the rts signal (n can be 0 or 1)

None of the above commands returns a reply (so pszresult can be NULL).

USB tower commands

On the USB tower's devicecontrol implementation, the pszcommand is always as follows:

<command>[(parameters)]=<new value> or <?> to retrieve the current value

Here's a list of the commands supported by the USB port layer.

range	Values: long, medium or short
linkmode	Values: ir, vll or irc
mode	Like linkmode
endianness	Values: little or big
flush	(No values) Flushes the firmware/driver receive/transmit buffers. Parameters: [rx,][tx]
reset	(No values) Resets the tower
timeout	Sets/retrieves the driver timeouts, in milliseconds. Parameters: ([read][read_1][read_ic][write])
version	(read-only) Retrieves the driver and firmware version and build number
copyright	(read-only) Retrieves a copyright information string
errorstate	(read-only) Retrieves the tower firmware internal error state (0==no errors)
ledmode	Sets/retrieves the current led mode (hw or software control). Values: hw or sw
ledstate	Values: on or off. Parameters: Led = **id** (the green led) or **vll** (the red led). Sets/retrieves the current led state (if the mode is set to software control).
irspeed	[(tx or rx)] Sets/retrieves the ir communication speed of the tower. If no arguments are specified, it sets both speeds to the same value, otherwise you can specify tx or rx in the parameter, and send two commands if you need to set different tx

Example

Serial tower

```
  PBKRESULT pbkrc;
  pbkrc = GhDeviceControl( hstack,"baud=2400 parity=O data=8
stop=1",NULL,0);
  pbkrc = GhDeviceControl( hstack,"dtr=1");
```

USB Tower

```
  PBKRESULT pbkrc;
  char szrange[MAXRANGESIZE];
  pbkrc = GhDeviceControl( hstack,"range=?",szrange,sizeof(szrange)
);
  pbkrc = GhDeviceControl( hstack,"mode=vll",NULL,0);
```

2.1.4.2 GhGetInterleave

Retrieve the currently selected interleave between the execute and download queue

Declaration

```
PBKRESULT   PBK_EXPORT GhGetInterleave(     GHSTACK     hstack,
                                            uint32
*pulinterleave_execute,
                                            uint32
*pulinterleave_download )
```

Parameters

hstack	(IN) Stack handle
pulinterleave_execute	(OUT) ptr to number of immediate command blocks
pulinterleave_download	(OUT) ptr to number of download slices

Return Value
PBKOK or errorcode

Remarks
See GhSetInterleave for explanation and examples of the parameters

See Also
GhSetInterleave

2.1.4.3 GhGetRetries

Get the number of retries for execute and download commands

Declaration

```
PBKRESULT   PBK_EXPORT GhGetRetries(    GHSTACK          hstack,
                                        int32
*plretries_execute,
                                        int32
*plretries_download )
```

Parameters

hstack	(IN) Stack handle
plretries_execute	(OUT) Ptr to current number of retries for immediate commands (GhExecute)
plretries_download	(OUT) Ptr to current number of retries for download slices (GhDownload)

Return Value
PBKOK or errorcode

Remarks
In order to be able to call this function, the stack has to be opened. It is possible to pass NULL as one of the parameters to this functions: e.g., if the caller is only interested in the download retries, he can call the function like this: GhSetRetries(hstack, NULL, &lretries_dl);

GhSetRetries, GhDownload, GhExecute

2.1.4.4 GhGetTimeout

Get current communication timeout values

Declaration

```
PBKRESULT   PBK_EXPORT GhGetTimeout(    GHSTACK      hstack,
                                        uint32      *pultimeout_1st,
                                        uint32      *pultimeout_ic
                                   )
```

Parameters

hstack	(IN) Stack handle
pultimeout_1st	(OUT) Ptr to 1st character timeout (in milliseconds)
pultimeout_ic	(OUT) Ptr to intercharacter timeout (in milliseconds)

Return Value

PBKOK or errorcode

Remarks

In order to be able to call this function, the stack has to be opened

See Also

GhGetTimeout

2.1.4.5 GhSetInterleave

Set the current command interleave between the execute and download queue

Declaration

```
PBKRESULT   PBK_EXPORT GhSetInterleave(     GHSTACK     hstack,
                                        uint32
ulinterleave_execute,
                                        uint32
ulinterleave_download    )
```

Parameters

hstack	(IN) Stack handle
ulinterleave_execute	(IN) number of immediate command blocks
ulinterleave_download	(IN) number of download slices

Return Value

PBKOK or errorcode

Remarks

this method lets you specify how immediate commands and download slices are interleaved by the session layer. Here are a few samples:

GhSetInterleave(hstack,1,1) sends one immediate command and one download slice.

GhSetInterleave(hstack,0,1) sends no immediate commands and one download slice. It means that immediate commands are sent only if there are no download slices queued.

GhSetInterleave(hstack,1,0) sends no download slices and one immediate command. It means that download slices are sent only if there are no immediate commands queued.

GhSetInterleave(hstack,2,3) sends two immediate commands followed by three download slices.

See Also
GhGetInterleave

2.1.4.6 GhSetNoWaitMode

Set the current notification mode to NOWAIT

Declaration

```
PBKRESULT   PBK_EXPORT GhSetNoWaitMode(     GHSTACK
hstack,
                                            GHNOTIFYFUNCTION
pfnotify    )
```

Parameters

hstack (IN) Stack handle

Return Value
PBKOK or errorcode (PBK_ERR_BADPARM, PBK_ERR_BUSY)

Remarks
In order to be able to call this function, the stack has to be opened and idle (i.e., no commands are currently being carried out).

Unlike GhSetWaitMode, GhSetNoWaitMode doesn't allow pfnnotify to be NULL (you have to have a notificatio function when working in NOWAIT mode).

See Also
GhSetWaitMode

2.1.4.7 GhSetRetries

Set the number of retries for execute and download commands

Declaration

```
PBKRESULT   PBK_EXPORT GhSetRetries(    GHSTACK             hstack,
                                        int32
lretries_execute,
                                        int32
lretries_download   )
```

Parameters

hstack (IN) Stack handle

lretries_execute (IN) Number of retries for immediate commands (GhExecute)

lretries_download (IN) Number of retries for download slices (GhDownload)

Return Value
PBKOK or errorcode

Remarks
In order to be able to call this function, the stack has to be opened; the new values will take effect with the next command to be submitted

GhGetRetries, GhDownload, GhExecute

2.1.4.8 GhSetTimeout

Set new communication timeout values

Declaration

```
PBKRESULT   PBK_EXPORT GhSetTimeout(    GHSTACK     hstack,
                                        uint32      ultimeout_1st,
                                        uint32      ultimeout_ic
                                    )
```

Parameters

hstack (IN) Stack handle

ultimeout_1st (IN) 1st character timeout (in milliseconds)

ultimeout_ic (IN) INtercharacter timeout (in milliseconds)

Return Value
PBKOK or errorcode

Remarks
In order to be able to call this function, the stack has to be opened; the new values will take effect with the next command to be submitted

See Also
GhGetTimeout

2.1.4.9 GhSetWaitMode

Set the current notification mode to WAIT

Declaration

```
PBKRESULT   PBK_EXPORT GhSetWaitMode(   GHSTACK hstack,
                                        GHNOTIFYFUNCTION    pfnotify
)
```

Parameters

hstack (IN) A valid stack handle

Return Value
PBKOK or errorcode (PBK_ERR_BUSY, PBK_ERR_BADPARM)

Remarks
In order to be able to call this function, the stack has to be opened and idle (i.e., no commands are currently being carried out).

If you want, you can pass a notification function pointer even when setting the WAIT mode. This means that your notification function will be called (on a different thread) before GhExecute or GhDownload return. If you don't want to have a notification function, you can pass NULL as pnotify.

GhSetNoWaitMode

2.1.5 Diagnostic Functions

2.1.5.1 GhDiagnose

Executes diagnostics functions to see if the device is currently attached, etc.

Declaration

```
PBKRESULT   PBK_EXPORT GhDiagnose( GHSTACK      hstack  )
```

Parameters

hstack (IN) Stack handle

Return Value

PBKOK or error code

2.1.5.2 GhDisableStatistics

Disable statistics collections

Declaration

```
PBKRESULT   PBK_EXPORT GhDisableStatistics( GHSTACK     hstack )
```

Parameters

hstack (IN) Stack handle

Return Value

PBKOK or error code

Remarks

Transmission statistics are kept by the Ghost's protocol layer. Statistics are disabled by default.

See Also

GhEnableStatistics, GhGetStatistics, GhResetStatistics

2.1.5.3 GhEnableStatistics

Enable statistics collections

Declaration

```
PBKRESULT   PBK_EXPORT GhEnableStatistics(  GHSTACK     hstack )
```

Parameters

hstack (IN) Stack handle

Return Value

PBKOK or error code

Remarks

Transmission statistics are kept by the Ghost's protocol layer. Statistics are disabled by default.

See Also
GhDisableStatistics, GhGetStatistics, GhResetStatistics

2.1.5.4 GhGetStatistics

Retrieve statistic values

Declaration

```
PBKRESULT   PBK_EXPORT GhGetStatistics( GHSTACK     hstack,
                                        uint32      *pulcmdreq,
                                        uint32      *pulcmdfail,
                                        uint32      *pulcmdabort,
                                        uint32      *pultxreq,
                                        uint32      *pultxfail,
                                        uint32      *pulrxreq,
                                        uint32      *pulrxfail
                                       )
```

Parameters

hstack	(IN) Stack handle
pulcmdreq	(OUT) Ptr to the number of total command requests
pulcmdfail	(OUT) Ptr to the number of total commdands failed
pulcmdabort	(OUT) Ptr to the number of total commdands aborted
pultxreq	(OUT) Ptr to the number of transmission attempts (>= ulcmdreq)
pultxfail	(OUT) Ptr to the number of transmission errors
pulrxreq	(OUT) Ptr to the number of receive requests
pulrxfail	(OUT) Ptr to the number of receive errors

Return Value
PBKOK or error code

Remarks
Any of the OUT parameter can be NULL (i.e., you can ask only for those statistic values you are interested in.

You can reset these values by calling GhResetStatistics.

See Also
GhEnableStatistics, GhDisableStatistics, GhResetStatistics

2.1.5.5 GhResetStatistics

Reset statistic counters

Declaration

```
PBKRESULT   PBK_EXPORT GhResetStatistics( GHSTACK       hstack  )
```

Parameters

hstack (IN) Stack handle

Return Value
PBKOK or error code

Remarks
This function resets to zero all the statistic counters returned by GhGetStatistics

See Also
GhEnableStatistics, GhDisableStatistics, GhGetStatistics

2.1.6 Type Definitions

2.1.6.1 GHCOMMAND

Handle to a single command. You get it by calling GhGetFirstCommand or GhGetNextCommand on a queue handle. You use a command handle to get specific information on the command (e.g., the reply len and data).

Declaration

```
typedef  void   *   GHCOMMAND
```

Example

```
uint32 PBK_STDCALL MyNotifyFunction(  GHNOTIFYCODE  notifycode, PBKRESULT
pbkresult, GHSTACK  hstack, GHQUEUE  hqueue, GHCOMMAND
hcommand, void  *pvcontext  ) { uint32 ulretcode = 1;

if ( notifycode == NotifyCommand) { uchar szcmd[256], szreply[256]; uint32 ulcmdlen,
ulreplylen; PBKRESULT pbkrc;


  pbkrc = GhGetCommandDataLen( hcommand, &ulcmdlen );
  ASSERT( PBK_SUCCEEDED(pbkrc) );
  pbkrc = GhGetCommandData( hcommand, szcmd, sizeof(szcmd) );
  ASSERT( PBK_SUCCEEDED(pbkrc) );
  pbkrc = GhGetCommandReplyLen( hcommand, &ulreplylen );
  ASSERT( PBK_SUCCEEDED(pbkrc) );
  if ( ulreplylen > 0 )
  {
      pbkrc = GhGetCommandReply( hcommand, szreply, sizeof(szreply)
);
      ASSERT( PBK_SUCCEEDED(pbkrc) );
  }
  printf("NotifyCommand: ");
  if ( ulcmdlen > 0 )
      printf("Cmd code: 0x%X ", (uint32)*szcmd );
  if ( PBK_SUCCEEDED(pbkresult))
  {
      printf("Succeeded ");
      if ( ulreplylen > 0 )
          printf("(Reply: 0x%X)", (uint32)*szreply );
  }
  else
  {
      printf("Failed (error code = 0x%X)", pbkresult );
// Cause the next commands to be aborted ulretcode = 0; } } return ulretcode; }
```

2.1.6.2 GHNOTIFYCODE

Notification codes. These codes are passed by the Ghost API to the notification function, in the ulnotify parameter.

Declaration

```
typedef enum {

            NotifyIgnore    = 0x00,
            NotifyCommand   = 0x03,
            NotifyRequest   = 0x04

    }GHNOTIFYCODE
```

2.1.6.3 GHNOTIFYFUNCTION

Declaration

```
typedef uint32 ( PBK_STDCALL *GHNOTIFYFUNCTION)(    GHNOTIFYCODE
ulnotifycode,
                                                        PBKRESULT
pbkresult,
                                                        GHSTACK
hstack,
                                                        GHQUEUE
hqueue,
                                                        GHCOMMAND
hcommand,
                                                        void
*pvcontext          )
```

Remarks
Callback function

[TODO] decide whether to keep the STDCALL modifier.

2.1.6.4 GHQUEUE

Handle to a command queue. You obtain this by calling GhCreateCommandQueue.

Declaration

```
typedef  void   *   GHQUEUE
```

Remarks
Once you have a queue handle, you can add commands to a queue (via GhAppendCommand). Then you can transmit all the commands in the queue by calling either GhDownload or GhExecute.

2.1.6.5 GHSTACK

Stack handle. You obtain this by calling GhCreateStack

Declaration

```
typedef  void   *   GHSTACK
```

Remarks
Most functions in the Ghost API take a stack handle as input parameter. When you don't need it any more, remember to release the stack by calling GhDestroyStack.

2.1.6.6 PBKRESULT

Declaration

```
typedef long PBKRESULT
```

Remarks
@type PBKRESULT | The general result type for the PBrick Comm API

2.1.6.7 int16

Declaration

```
typedef signed short int16
```

Remarks
@type int16 | 16-bit signed value

2.1.6.8 int32

Declaration

```
typedef signed long int32
```

Remarks
@type int32 | 32-bit signed value

2.1.6.9 int8

Declaration

```
typedef signed char int8
```

Remarks
@type int8 | 8-bit signed integer

2.1.6.10 uchar

Declaration

```
typedef unsigned char uchar
```

Remarks
@type uchar | unsigned char

2.1.6.11 uint

Declaration

```
typedef unsigned int uint
```

Remarks
@type uint | unsigned int

2.1.6.12 uint16

Declaration

```
typedef unsigned short uint16
```

Remarks
@type uint16 | unsigned short (16 bits)

2.1.6.13 uint32

Declaration

```
typedef unsigned long uint32
```

Remarks
@type uint32 | unsigned long (32 bits)

2.1.6.14 uint8

Declaration

```
typedef unsigned char uint8
```

Remarks
@type uint8 | 8-bit unsigned integer (same as uchar)

2.1.7 External Functions

2.1.7.1 GhDownloadEx

Declaration

```
PBKRESULT   PBK_EXPORT GhDownloadEx(    GHSTACK     hstack,
                                        GHQUEUE     hqueue,
                                        void        *pcontext )
```

Remarks
Submit a command queue on the DOWNLOAD queue

2.1.7.2 GhExecuteEx

Declaration

```
PBKRESULT   PBK_EXPORT GhExecuteEx(     GHSTACK     hstack,
                                        GHQUEUE     hqueue,
                                        void        *pcontext )
```

Remarks
Submit a command queue on the EXECUTE queue passing in a specific context value

2.1.7.3 PBK_FAILED

Declaration

```
bool PBK_FAILED(PBKRESULT e)
```

2.1.7.4 PBK_SUCCEEDED

Declaration

```
bool PBK_SUCCEEDED(PBKRESULT e)
```

2.1.8 Macros

2.1.8.1 FAC_PBK

Declaration

```
#define FAC_PBK                         0x01000000
```

Remarks

error codes generated in the Ghost communication stack (session, protocol, port)

2.1.8.2 FAC_USBTOWER

Declaration

```
#define FAC_USBTOWER                    0x02000000
```

Remarks
error codes from the USB Tower firmware are translated into Win32 errors using this facility

2.1.8.3 PBKCOMM_EXPORT

Declaration

```
#define PBKCOMM_EXPORT  __declspec(dllexport)
```

Remarks
Obsolete - use only for backward compatibility

2.1.8.4 PBKERR_CUSTOM

Declaration

```
#define PBKERR_CUSTOM                   0x20000000
```

2.1.8.5 PBKERR_MAKE

Declaration

```
#define PBKERR_MAKE( sev, fac, code )   ( sev | PBKERR_MASK_FAC(fac)
| code )
```

2.1.8.6 PBKERR_MAKE_ERROR

Declaration

```
#define PBKERR_MAKE_ERROR( fac, code )  (PBKERR_MAKE(
PBKERR_SEVERITY_ERROR, fac , code ) )
```

2.1.8.7 PBKERR_MAKE_INFO

Declaration

```
#define PBKERR_MAKE_INFO( fac, code )   (PBKERR_MAKE(
PBKERR_SEVERITY_INFORMATIONAL, PBKERR_MASK_FAC(fac), code ) )
```

2.1.8.8 PBKERR_MAKE_SUCCESS

Declaration

```
#define PBKERR_MAKE_SUCCESS( fac, code )(PBKERR_MAKE(
PBKERR_SEVERITY_SUCCESS, PBKERR_MASK_FAC(fac), code ) )
```

2.1.8.9 PBKERR_MAKE_WARNING

Declaration

```
#define PBKERR_MAKE_WARNING( fac, code )(PBKERR_MAKE(
PBKERR_SEVERITY_WARNING, PBKERR_MASK_FAC(fac), code ) )
```

2.1.8.10 PBKERR_MASK

Declaration

```
#define PBKERR_MASK                     (PBKERR_CUSTOM | FAC_PBK )
```

2.1.8.11 PBKERR_MASK_ERROR

Declaration

```
#define PBKERR_MASK_ERROR               (PBKERR_SEVERITY_ERROR
| PBKERR_MASK)
```

2.1.8.12 PBKERR_MASK_FAC

Declaration

```
#define PBKERR_MASK_FAC(fac)            (PBKERR_CUSTOM | fac )
```

2.1.8.13 PBKERR_MASK_INFO

Declaration

```
#define PBKERR_MASK_INFO
(PBKERR_SEVERITY_INFORMATIONAL  | PBKERR_MASK)
```

2.1.8.14 PBKERR_MASK_SUCCESS

Declaration

```
#define PBKERR_MASK_SUCCESS             (PBKERR_SEVERITY_SUCCESS
| PBKERR_MASK)
```

2.1.8.15 PBKERR_MASK_WARNING

Declaration

```
#define PBKERR_MASK_WARNING             (PBKERR_SEVERITY_WARNING
| PBKERR_MASK)
```

2.1.8.16 PBKERR_SEVERITYBITS

Declaration

```
#define PBKERR_SEVERITYBITS           0xC0000000
```

2.1.8.17 PBKERR_SEVERITY_ERROR

Declaration

```
#define PBKERR_SEVERITY_ERROR         0xC0000000
```

2.1.8.18 PBKERR_SEVERITY_INFORMATIONAL

Declaration

```
#define PBKERR_SEVERITY_INFORMATIONAL 0x40000000
```

2.1.8.19 PBKERR_SEVERITY_SUCCESS

Declaration

```
#define PBKERR_SEVERITY_SUCCESS       0x00000000
```

2.1.8.20 PBKERR_SEVERITY_WARNING

Declaration

```
#define PBKERR_SEVERITY_WARNING       0x80000000
```

2.1.8.21 PBKOK

Declaration

```
#define PBKOK       0
```

Remarks
@const | PBKOK | No error occurred

2.1.8.22 PBK_EXPORT

Declaration

```
#define PBK_EXPORT      __declspec(dllexport)
```

2.1.8.23 PBK_IS_ERROR

Declaration

```
#define PBK_IS_ERROR(e)
((e&PBKERR_SEVERITYBITS)==PBKERR_SEVERITY_ERROR)
```

2.1.8.24 PBK_IS_INFO

Declaration

```
#define PBK_IS_INFO(e)
((e&PBKERR_SEVERITYBITS)==PBKERR_SEVERITY_INFORMATIONAL)
```

2.1.8.25 PBK_IS_SUCCESS

Declaration

```
#define PBK_IS_SUCCESS(e)
((e&PBKERR_SEVERITYBITS)==PBKERR_SEVERITY_SUCCESS)
```

2.1.8.26 PBK_IS_WARNING

Declaration

```
#define PBK_IS_WARNING(e)
((e&PBKERR_SEVERITYBITS)==PBKERR_SEVERITY_WARNING)
```

2.1.8.27 PBK_STDCALL

Declaration

```
#define PBK_STDCALL         __stdcall
```

Remarks
Definition for the stdcall ("pascal") calling convention.

2.1.8.28 USBTOWER_ERR_MAX

Declaration

```
#define USBTOWER_ERR_MAX        255
```

2.1.8.29 USBTOWER_ERR_MIN

Declaration

```
#define USBTOWER_ERR_MIN        1
```

2.1.8.30 _WIN32

Declaration

```
#define _WIN32
```

2.1.8.31 __MACINTOSH__

Declaration

```
#define __MACINTOSH__
```