PROFIBUS Application Program Interface

User Interface

Version 5.4
Rev. 07

Date: 14-October-2011

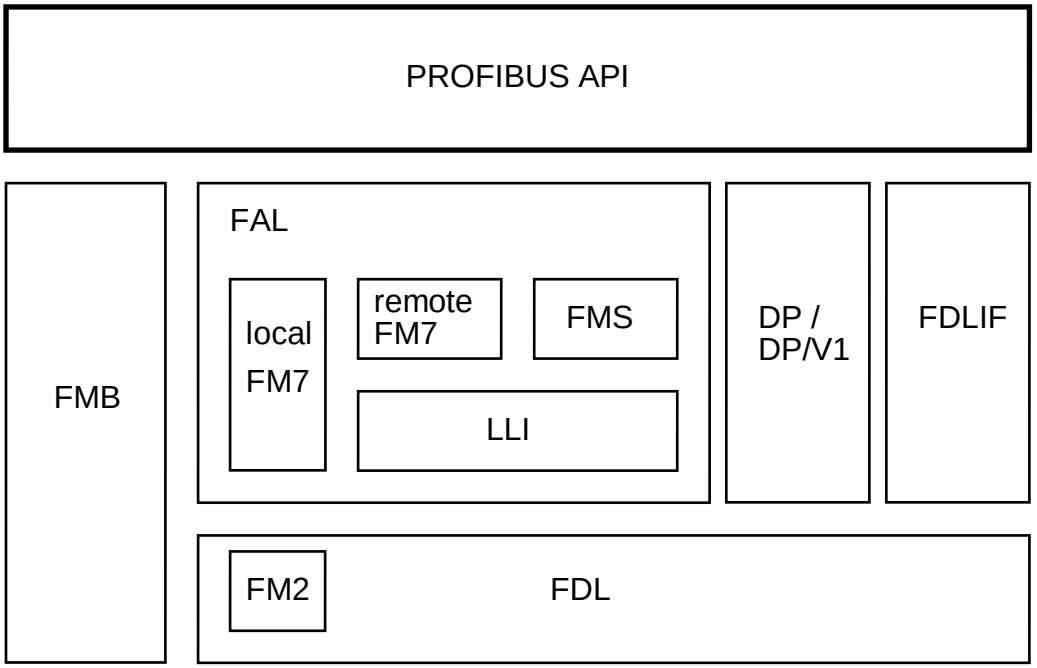
Softing Industrial Automation GmbH
Richard-Reitzner-Allee 6
D-85540 Haar
Phone (++49) 89 - 4 56 56-0
Fax (++49) 89 - 4 56 56-399

Copyright Notice

CONTENTS

1 SCOPE

This manual describes the common access functions to all components of Softing's PROFIBUS protocol software .



Depending on the component of the PROFIBUS protocol to be used, this document should be read in conjunction with one or more of the following parts of the PROFIBUS User Manual:

- "Basic Management"
- "FMS Services"
- "FM7 Services"
- "DP Services"
- "DP/V1 Services"
- "DP-Slave Services"
- "FDL Services"

2 OVERVIEW

The services the PROFIBUS protocol (FMS, FM7 and DP / DP/V1) offers to the user are described in detail in IEC 61158-5 AND IEC 61158-6 .

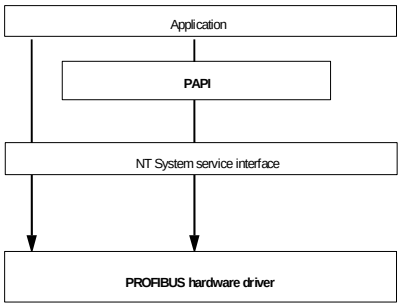
Those references describe the functions and parameters of the management and communication services, but do not provide instructions or structures for the programmer on the nuts and bolts of how to write an interface. This leaves room for implementations which can be adapted to optimally fit their respective system environment on the positive side, but which can also reduce the "openness" of the application layer interfaces.

3 USER INTERFACE USING WINDOWS 2000 AND HIGHER

The PROFIBUS driver for Windows 2000 and higher offers access to the functionality of the PROFIBUS protocol stack which runs on the PC boards PROFIboard-PCI, PROFIcard 2, PROFIusb, PROFI104, PBpro-PCI, PBpro-cPCI, PBpro-PCIe, PBpro-PC104+ and on the PROFIBUS-Ethernet gateways PBpro ETH and FG-x00-PB.

The software consists of the following parts:

- A kernel device driver, the PROFIBUS hardware driver, which provides access to the plugged PROFIBUS interfaces and remote access via Ethernet to PBpro ETH / FG-x00-PB.
- An interface library (PAPI - PROFIBUS Application Program Interface) which provides access to the complete functionality of the PROFIBUS hardware driver, and, in addition, offers a compatibility mode interface to simplify porting of existing PROFIBUS applications to Windows.
- An .NET interface library (PAPI - PROFIBUS Application Program Interface) which also provides access to the complete functionality of the PROFIBUS hardware driver.



3.1 LOGICAL DEVICES

The PROFIBUS hardware device driver creates the logical devices during system startup.

The logical devices are accessed by the I/O functions of Windows (32Bit / 64Bit). Each logical device must be opened before it can be used. The devices are accessed by read, write, and control functions. After using a device, it should always be closed.

All read requests can be executed only if the device is open for read access. All write requests can be executed only if the device is open for write access. The access rights needed for control functions depend on the specific control function code.

3.1.1 Directory structure of logical devices

To access a logical device, a unique Windows device name (Symbolic Link Name) is required. Optionally a Windows alias device name can be defined via the PROFIBUS control panel. The device names are created by the kernel device drivers.

Note: The Windows subsystem requires that all device names begin with the characters "\\.\".

The logical devices are structured hierarchically similar to a directory structure:

Example to access the **general service device** of board 0:

- Access via Windows device name: \\.\PROFIBUS\Board0\Pb0\Service
- Access via Windows alias device name: \\.\PROFIBUS\Node0\Service

NOTES:

In C programs, each backslash must be typed as "\\". Consequently, this device name in a C program is "\\\\.\\PROFIBUS\\Board0\\Pb0\\Service " or "\\\\.\\PROFIBUS\\Node0\\Service".

The device names of all devices described below use these variable:

***Y* is the board number**

3.1.2 Devices

3.1.2.1 Board device

Windows Name: \\.\PROFIBUS\Board*Y*\Board
Windows Alias Name: \\.\PROFIBUS\<SymbolicNodeName>\Board

Function: Services concerning the complete board

Read: Read the version information of the PROFIBUS protocol firmware
Ioctl: IOCTL_PROFI_GET_DATA_IMAGE
Get a data image from the board

3.1.2.2 General service device

Windows Name: \\.\PROFIBUS\Board*Y*\Pb0\Service
Windows Alias Name: \\.\PROFIBUS\<SymbolicNodeName>\Service

Function: Read and write any PROFIBUS frame

Read: Read any received frame
Write: Write any frame
Ioctl: IOCTL_PROFI_SET_TIMEOUT
IOCTL_PROFI_GET_TIMEOUT
Set or read the time-outs for read/write operations to/from this device.
'Set' requires read access to the device.
'Get' can be done with any access to the device
The default time-out values of the General Service Device are 0 ms for read and write.

If writing of a frame fails with one of these error codes (`E_IF_NO_PARALLEL_SERVICES`, `E_IF_RESOURCE_UNAVAILABLE`, `E_IF_SERVICE_CONSTR_CONFLICT`), the hardware driver retries to write the frame until either the write succeeds or the write time-out elapsed.

NOTE:

The FMB_EXCEPTION indication will be generated by the firmware if a fatal error has been detected. The PROFIBUS hardware will be reinitialized.

If a FMB_RESET or FMB_EXIT confirmation has been received, the PROFIBUS hardware will be reinitialized.

3.1.2.3 General DP-Master data device

Windows Name: \\.\PROFIBUS\Board*Y*\Pb0\DpData
Windows Alias Name: \\.\PROFIBUS\<SymbolicNodeName>\DpData

Function: Read and write of any DP data

Read: Read any DP data, not restricted to a slave
Write: Write any DP data, not restricted to a slave
Ioctl: Set / Clear DP data bits in I/O data image, not restricted to a slave

3.1.2.4 General DP-Slave input data device

Windows Name: \\.\PROFIBUS\Board*Y*\Pb0\DpSlaveInputData
Windows Alias Name: \\.\PROFIBUS\<SymbolicNodeName>\DpSlaveInputData

Function: Read and write of any DP-Slave input data

Read: Read DP-Slave input data and current status
Ioctl: Write DP-Slave input data, read current status

3.1.2.5 General DP-Slave output data device

Windows Name: \\.\PROFIBUS\Board*Y*\Pb0\DpSlaveOutputData
Windows Alias Name: \\.\PROFIBUS\<SymbolicNodeName>\DpSlaveOutputData

Function Read DP-Slave output data

Read: Read DP-Slave output data and current status

3.1.3 Access rights

The board device may be opened any times for read access (to read the firmware version information), but only once for write access. It may be opened for write access only if there is no other device open on this board.

The service-oriented General Service Device may be opened any times for write access, but only once for read access.

All data-oriented devices (General Data Device, DP Slave Data Devices) may opened any times for read access, but only once for write access.

3.2 PROGRAM INTERFACES

The PROFIBUS API for Windows supports two different program interfaces. Each of these interfaces has its own advantages (+) and disadvantages (-). The following list should help you decide which interface to use.

- **Windows System Interface**

  The Windows system interface consists of the standard Windows system calls for device handling. These functions can be used to access the devices provided by the low-level kernel mode device driver.

  + Provides the whole functionality of the device drivers

  + Asynchronous read or write calls possible

  – Most difficult to program

- **PROFIBUS Application Program Interface**

  The PROFIBUS Application Program Interface (PAPI) provides two mechanisms for data exchange between application and protocol software and host and controller:

  - a send/receive interface using request blocks for service-oriented data exchange and

  - a data interface, which is used for fast cycle data exchange.

  All PROFIBUS API calls are implemented based on the functionality provided by the low-level device driver.

3.3 PROFIBUS WINDOWS SYSTEM INTERFACE

The following Windows system calls can be used to handle the PROFIBUS devices. These calls are subsequently described in detail:

CreateFile	Open a PROFIBUS device
CloseHandle	Close a PROFIBUS device
GetLastError	Get the error code of the last failed system call
DeviceIOControl	Send a control code to a PROFIBUS device
ReadFile	Read data from a PROFIBUS device
ReadFileEx	Read data asynchronously from a PROFIBUS device
WriteFile	Write data to a PROFIBUS device
WriteFileEx	Write data asynchronously to a PROFIBUS device
SetFilePointer	Set the file pointer of a general data device
FileIOCompletionRoutine	Callback routine for asynchronous data transfer

3.3.1 Data structures

All Windows system Interface provides access to the service-oriented devices. A PROFIBUS frame consists of a service-independent description and a service-specific service-specific data block with parameters and data.

The data structure T_PROFI_SERVICE_DESCR describes the service to be performed by the protocol software.

Description of the service description block:

```
typedef struct T_PROFI_SERVICE_DESCR
        {
        USIGN16             comm_ref;
        USIGN8              layer;
        USIGN8              service;
        USIGN8              primitive;
        INT8                invoke_id;
        INT16               result;
        } T_PROFI_SERVICE_DESCR;
```

The service description block's elements are as follows:

- comm_ref : Communication reference ("logical channel")
- layer: Layer instance the service invocation is directed to (FMS, FMB, FM7, DP, DPS, FDLIF, FMS_USR, FMB_USR, FM7_USR, DP_USR, DPS_USR, FDLIF_USR)
- service_id : Service to be performed in the instance specified in the layer.
- primitive : Service primitives (request, indication, response, confirmation)
- invoke_id : ID to distinguish parallel service invocations
- result : Positive or negative result

The data block contains the service-specific data. Typically, for communication services these are data as described in the PROFIBUS IEC 61158-5 AND IEC 61158-6

Construction of the service-specific data blocks is described in the manuals FMS, FMB, FM7, FDLIF, DP and DP/V1.

3.3.1 CreateFile

The **CreateFile** function opens a PROFIBUS device and returns a handle that can be used to access the object.

```
HANDLE CreateFile
    (
    LPCTSTR                 lpFileName,
    DWORD                   dwDesiredAccess,
    DWORD                   dwShareMode,
    LPSECURITY_ATTRIBUTES   lpSecurityAttributes
    DWORD                   dwCreationDistribution,
    DWORD                   dwFlagsAndAttributes,
    HANDLE                  hTemplateFile
    );
```

Function parameter description:

lpFileName: Points to a null-terminated string that specifies the name of the PROFIBUS device to open.

dwDesiredAccess: Specifies the type of access to the device. An application can obtain read access, write access or read-write access. Use the following flag constants to build a value for this parameter. Both GENERIC_READ and GENERIC_WRITE must be set to obtain read/write access. If dwDesiredAccess is 0, neither read nor write access is allowed; only IOControl operations that do not need a specific access right can be performed on the device.

Value	Meaning
GENERIC_READ	Specifies read access to the device. Data can be read from the device, and the file pointer can be moved.
GENERIC_WRITE	Specifies write access to the device. Data can be written to the device, and the file pointer can be moved.

dwShareMode: Set of bit flags that specifies how the device can be shared. If dwShareMode is 0, the device cannot be shared. No other open operations can be performed on the device. This flag cannot extend the constraints described in chapter 4.1.6 (Access rights).To share the device, use a combination of one or more of the following values:

Value	Meaning
FILE_SHARE_READ	Other open operations can be performed on the device for read access.
FILE_SHARE_WRITE	Other open operations can be performed on the device for write access.

lpSecurityAttributes: Always should be NULL; security is not supported by the PROFIBUS device drivers.

dwCreationDistribution: Must be OPEN_EXISTING.

dwFlagsAndAttributes: Specifies the file attributes and flags for the file.

Attribute	Meaning
FILE_ATTRIBUTE_NORMAL	The file has no other attributes set. This attribute is valid only if used alone.
FILE_FLAG_OVERLAPPED	Instructs the operating system to initialize the device so **ReadFile** and **WriteFile** operations that take a significant amount of time to process return ERROR_IO_PENDING. When the operation is finished, an event is set to the signaled state. When you specify FILE_FLAG_OVERLAPPED, the **ReadFile** and **WriteFile** functions must specify an **OVERLAPPED** structure: i.e. when FILE_FLAG_OVERLAPPED is specified, an application must perform overlapped reading and writing. General Data Device: When FILE_FLAG_OVERLAPPED is specified, the file position must be passed as part of the lpOverlapped parameter (pointing to an **OVERLAPPED** structure) to the **ReadFile** and **WriteFile** functions. This flag also enables more than one operation to be performed simultaneously with the handle (a simultaneous read and write operation, for example).

hTemplateFile: This value must be NULL.

Possible function return values

- If the function succeeds, the return value is an open handle to the specified device.
- If the function fails, the return value is INVALID_HANDLE_VALUE. To obtain extended error information, call **GetLastError**.

NOTES:

You can use the CreateFile function to open a logical PROFIBUS device. The function returns a handle to the device. This handle can be used with the ReadFile, WriteFile, and DeviceIOControl function.

The CloseHandle function is used to close a handle returned by CreateFile.

Example

```
#include <windows.h>
...
{
  HANDLE hBoard;
  ULONG  ErrorCode;

  hBoard = CreateFile   ("\\\\.\\PROFIBUS\\Board0\\Board"          // Name of the device
                         GENERIC_READ,                                    // Access mode
                         FILE_SHARE_READ                                   // Share mode
                         NULL,                           // Pointer to securitydescriptor
                         OPEN_EXISTING,                                  // How to create
                         FILE_ATTRIBUTE_NORMAL,                    // File attribute
                         NULL,                                  // Handle to template file
                        );

  if (hBoard == INVALID_HANDLE_VALUE)
  {
     // do error handling
     ErrorCode = GetLastError();
     ...
  }
  ...
}
```

3.3.2 CloseHandle

The **CloseHandle** function closes an open handle.

```
BOOL CloseHandle
        (
        HANDLE          hObject
        );
```

Function parameter description:

hObject: Identifies an open device handle.

Possible function return values

- If the function succeeds, the return value is **TRUE**.-
- If the function fails, the return value is **FALSE**. To obtain extended error information, call **GetLastError**.

NOTES:

CloseHandle invalidates the specified object.

Use CloseHandle to close handles returned by calls to the CreateFile function.

Closing an invalid handle raises an exception. This includes closing a handle twice and not checking the return value and closing an invalid handle.

Example

```
{
  HANDLE hDevice;

  // Open device
  ULONG  ErrorCode;

  hDevice = CreateFile (....

  // do input / output
  ...

  // close device
  if (!CloseHandle (hDevice))
  {
    // do error handling
    ErrorCode = GetLastError();
    ...
  }

}
```

3.3.3 GetLastError

The **GetLastError** function returns the calling thread's last-error code value. The last-error code is maintained on a per-thread basis. Multiple threads do not overwrite each other's last-error code.

```
DWORD GetLastError
        (
        VOID
        );
```

Function parameter description:

This function has no parameters.

Possible function return values

The return value is the calling thread's last-error code value.

NOTES:

You should call the GetLastError function immediately when a return value of a function indicates that such a call will return useful data. Reason: Some functions call SetLastError(0) when they succeed, wiping out the error code set by the most recently failed function.

Most functions in the Windows API that set the thread's last error code value set it when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value error code such as FALSE, NULL, 0xFFFFFFFF, or –1. Some functions call SetLastError under conditions of success; these cases are noted in the reference page of each function.

The PROFIBUS driver functions only set the last error code when they fail.

Error codes are 32-bit values (bit 31 is the most significant bit). Bit 29 is called the "Customer code flag" and is reserved for application-defined error codes; no system error code has this bit set. This bit set to one indicates that the error code is defined by the PROFIBUS software. The lower two bytes include the PROFIBUS error code.

To obtain an error string for operating system error codes, use the FormatMessage function. A complete list of system error codes can be found in the WINERROR.H header file in the Windows SDK. The list of application-defined error codes can be found in this manual.

Example

```
#include <windows.h>
#include "pb_err.h"

#define CUSTOMER_CODE_FLAG 0x20000000
...
{
ULONG ErrorCode;

// do PROFIBUS IO
...

// get last error
ErrorCode = GetLastError ();

// check for Customer code flag
if (ErrorCode & CUSTOMER_CODE_FLAG)
{
  // Customer code flag is set: do profibus error handling
  // profibus error code is the low word of ErrorCode
  switch (LOWORD(ErrorCode))
  {
    case E_IF_FATAL_ERROR:
         ...
         break;

    case E_IF_SERVICE_CONSTRAINT_CONFLICT:
         ...
         break;
    ...
  }
}
else // System error
{
 ...
}

}
```

3.3.4 DeviceIoControl

The **DeviceIoControl** function sends a control code directly to a specified device driver, causing the corresponding logical device to perform the specified operation.

```
BOOL DeviceIoControl
    (
    HANDLE          hDevice,
    DWORD           dwIoControlCode,
    LPVOID          lpInBuffer,
    DWORD           nInBufferSize,
    LPVOID          lpOutBuffer,
    DWORD           nOutBufferSize,
    LPDWORD         lpBytesReturned,
    LPOVERLAPPED    lpOverlapped
    );
```

Function parameter description:

hDevice:	Handle to the device that is to perform the operation. Call the **CreateFile** function to obtain a device handle.
dwIoControlCode:	Specifies the control code for the operation. This value identifies the specific operation to be performed and the type of device on which the operation is to be performed. The values defined are described later in this section.
lpInBuffer:	Pointer to a buffer that contains the data required to perform the operation. This parameter can be NULL if the dwIoControlCode parameter specifies an operation that does not require input data.
nInBufferSize:	Specifies the size, in bytes, of the buffer pointed to by lpInBuffer.
lpOutBuffer:	Pointer to a buffer that receives the operation's output data. This parameter can be NULL if the dwIoControlCode parameter specifies an operation that does not produce output data.
nOutBufferSize:	Specifies the size, in bytes, of the buffer pointed to by lpOutBuffer.
lpBytesReturned:	Pointer to a variable that receives the size, in bytes, of the data stored into the buffer pointed to by lpOutBuffer. lpBytesReturned cannot be NULL. Even when an operation does not produce output data, and lpOutBuffer can be NULL, the **DeviceIoControl** function makes use of the variable pointed to by lpBytesReturned. After such an operation, the value of the variable is inapplicable.
lpOverlapped:	Pointer to an **OVERLAPPED** structure. If hDevice was opened with the FILE_FLAG_OVERLAPPED flag, this parameter must point to a valid **OVERLAPPED** structure. In this case, **DeviceIoControl** is performed as an overlapped (asynchronous) operation. If the device was opened with FILE_FLAG_OVERLAPPED and lpOverlapped is NULL, the function fails in unpredictable ways. If hDevice was opened without specifying the FILE_FLAG_OVERLAPPED flag, this parameter is ignored, and the **DeviceIoControl** function does not return until the operation has been completed or an error occurs.

Possible function return values

- If the function succeeds, the return value is **TRUE**.-
- If the function fails, the return value is **FALSE**. To obtain extended error information, call **GetLastError**.

NOTES:

If *hDevice* was opened with FILE_FLAG_OVERLAPPED and the *lpOverlapped* parameter points to an OVERLAPPED structure, DeviceIoControl is performed as an overlapped (asynchronous) operation. In this case, the OVERLAPPED structure must contain a handle to a manual-reset event object created by a call to the CreateEvent function.

If the overlapped operation cannot be completed immediately, the function returns FALSE, and GetLastError returns ERROR_IO_PENDING, indicating that the operation is executing in the background. When this happens, the operating system sets the event object in the OVERLAPPED structure to the non-signaled state before DeviceIoControl returns. The system then sets the event object to the signaled state when the operation has been completed. The calling thread can use any of the wait functions to wait for the event object to be signaled, and then use the GetOverlappedResult function to determine the results of the operation. The GetOverlappedResult function reports the success or failure of the operation and the number of bytes returned in the *lpOutBuffer* buffer.

Control codes

The following control codes are supported by the PROFIBUS device driver and defined in the file "pb_ntdrv.h":

Value	Meaning
IOCTL_PROFI_SET_TIMEOUT	Set time-out values for read and write lpInBuffer: Pointer to a PROFI_TIMEOUT data nInBufferSize: Size of PROFI_TIMEOUT data lpOutBuffer: NULL nOutBufferSize: 0 Devices: General service device Required Access: Read
IOCTL_PROFI_GET_TIMEOUT	Read time-out values for read and write lpInBuffer: NULL nInBufferSize: 0 lpOutBuffer Pointer to PROFI_TIMEOUT data nOutBufferSize Size of PROFI_TIMEOUT data Devices: General service device Required Access: None

Value	Meaning
IOCTL_PROFI_GET_DATA_IMAGE	Read a data image from the controller.
	lpInBuffer: Pointer to PROFI_DATA_IMAGE_DESCR data
	nInBufferSize: Size of the PROFI_DATA_IMAGE_DESCR data
	lpOutBuffer: Buffer to receive the data image
	nOutBufferSize: Size of the data image buffer
	lpBytesReturned: Size of the read date image
	Devices: General board device
	Required Access: None
IOCTL_PROFI_SET_DPS_DATA	Write DP-Slave input data and read DP-Slave input data state.
	lpInBuffer: DP-Slave input data
	nInBufferSize: Size of the DP-Slave input data
	lpOutBuffer: Buffer to receive the input data state
	nOutBufferSize: sizeof(USIGN8)
	lpBytesReturned: Size of the read data
	Devices: DP-Slave input data device
	Required Access: None
IOCTL_PROFI_SET_DP_BITS	Set Bits in DP slave I/O data image.
	lpInBuffer: Pointer to PROFI_DP_BIT_ACCESS data
	nInBufferSize: Size of PROFI_DP_BIT_ACCESS data
	lpOutBuffer: NULL
	nOutBufferSize: 0
	lpBytesReturned: Pointer to variable to receive output byte count, always zero
	Devices: General DP-Master data device
	Required Access: None
IOCTL_PROFI_CLEAR_DP_BITS	Clear Bits in DP slave I/O data image.
	lpInBuffer: Pointer to PROFI_DP_BIT_ACCESS data
	nInBufferSize: Size of PROFI_DP_BIT_ACCESS data
	lpOutBuffer: NULL
	nOutBufferSize: 0
	lpBytesReturned: Pointer to variable to receive output byte count, always zero
	Devices: General DP-Master data device
	Required Access: None

Example

```
{
  HANDLE hService;
  ULONG  nBytes;
  ULONG  ReadWriteTimeout[2];

  // Open service device
  hService = CreateFile ("\\\\.\\PROFIBUS\\Board0\\Pb0\\Service",
                         GENERIC_READ,0,NULL,OPEN_EXISTING,
                         FILE_ATTRIBUTE_NORMAL,NULL);

  if (hService != INVALID_HANDLE)
  {
     ReadWriteTimeout[0] = 100;
     ReadWriteTimeout[1] = 100;

     if (!DeviceIoControl((HANDLE) hService,
                          (DWORD)   IOCTL_PROFI_SET_TIMEOUT,
                          (LPVOID)  ReadWriteTimeout,
                          (DWORD)   2 * sizeof(ULONG),
                          (LPVOID)  NULL,
                          (DWORD)   0,
                          (LPDWORD) &nBytes,
                          NULL
      {
        // do error handling - DeviceIoControl failed
        ...
      )
      ...
  }
}
```

3.3.5 ReadFile

The **ReadFile** function reads data from a device. See the description of each logical device for a description of the data to read. File positions are considered only during read operations from the general data devices.

```
BOOL ReadFile
        (
        HANDLE          hFile,
        LPVOID          lpBuffer,
        DWORD           nNumberOfBytesToRead,
        LPDWORD         lpNumberOfBytesRead,
        LPOVERLAPPED    lpOverlapped
        );
```

Function parameter description:

hFile:	Identifies the file to be read. The file handle must have been created with GENERIC_READ access to the file. For asynchronous read operations, hFile can be any handle opened with the FILE_FLAG_OVERLAPPED flag by the **CreateFile** function.
lpBuffer:	Points to the buffer that receives the data read from the device.
nNumberOfBytesToRead:	Specifies the maximum number of bytes to be read from the device.
lpNumberOfBytesRead:	Points to the number of bytes read. If lpOverlapped is NULL, lpNumberOfBytesRead cannot be NULL. If lpOverlapped is not NULL, lpNumberOfBytesRead can be NULL. If this is an overlapped read operation, the number of bytes read can be fetched by calling **GetOverlappedResult**.
lpOverlapped:	Points to an **OVERLAPPED** structure. This structure is required if hFile was created with FILE_FLAG_OVERLAPPED. If hFile was opened with FILE_FLAG_OVERLAPPED, the lpOverlapped parameter must not be NULL. It must point to a valid **OVERLAPPED** structure. If hFile was created with FILE_FLAG_OVERLAPPED and lpOverlapped is NULL, the function can incorrectly report that the read operation is complete. If hFile was opened with FILE_FLAG_OVERLAPPED and lpOverlapped is not NULL, the read operation starts at the offset specified in the **OVERLAPPED** structure and **ReadFile** may return before the read operation has been completed. In this case, **ReadFile** returns FALSE, and the **GetLastError** function returns ERROR_IO_PENDING. This allows the calling process to continue while the read operation finishes. The event specified in the **OVERLAPPED** structure is set to the signaled state upon completion of the read operation. If hFile was not opened with FILE_FLAG_OVERLAPPED and lpOverlapped is NULL, the read operation starts at the current file position, and **ReadFile** does not return until the operation has been completed. If hFile is not opened with FILE_FLAG_OVERLAPPED and lpOverlapped is not NULL, the read operation starts at the offset specified in the **OVERLAPPED** structure. **ReadFile** does not return until the read operation has been completed.

Possible function return values:

- If the function succeeds, the return value is **TRUE**. If the return value is TRUE and the number of bytes read is zero, no data are available at the device.
- If the function fails, the return value is **FALSE**. To obtain extended error information, call **GetLastError**.

NOTES:

Applications must neither read from nor write to the input buffer that a read operation is using until the read operation completes. A premature access to the input buffer may lead to corruption of the data read into that buffer.

The ReadFile function may fail and return ERROR_INVALID_USER_BUFFER or ERROR_NOT_ENOUGH_MEMORY whenever there are too many outstanding asynchronous I/O requests.

If you read from the general service device, the comm_ref field of the service descriptor block contains a number of status bits in the high byte. Only the low byte contains the communication reference of the service.

Usage

Board device: Reads the version information of the PROFIBUS protocol firmware

lpBuffer:	pointer to an array of characters
nNumberOfBytesToRead:	the maximum size of the version information is defined in VERSION_STRING_LENGTH

Service-oriented device: Reads a received frame. The frame consists of the service description followed by the service and primitive specific data.

lpBuffer:	pointer to the buffer that receives the frame data
nNumberOfBytesToRead:	size of the buffer pointed by lpBuffer. A frame could have the maximum size defined in MAX_FMS_PDU_LENGTH
lpNumberOfBytesRead:	size of the received frame. If the size of the frame is 0 and the function succeeded, no frame was received during the time-out time.

Data-oriented devices: Reads DP data. The DP slave data devices check the status information of the slave an return the error `E_SLAVE_ERROR` if the slave status is bad.

lpBuffer:	Pointer to the buffer that receives the DP data
nNumberOfBytesToRead	Size of the buffer. The maximum amount of bytes to read on DP slave data devices is the maximum read size of the slave.

Examples

```
{
  HANDLE hBoard;                                        // Handle of the board device
  HANDLE hService;                                 // Handle of the general service device
  char   FirmwareVersion[VERSION_STRING_LENGTH];
  char   Data[255];

  // Open board and service device
  ...

  // Read the firmware version info from the board device
  if(ReadFile(hBoard,FirmwareVersion,VERSION_STRING_LENGTH,&nBytes,NULL))
  {
    // read version info
    ...
  }

  // Read from the service device
  if(ReadFile(hService,Data,255,&nBytes,NULL))
  {
    if (nBytes > 0)
      // Frame received
    else
      // No frame received during time-out

  ...
  }
}
```

```
{
  HANDLE hData;                                         // Handle of the DP data device
  BYTE   Data[255];

  // Create and open the DP data device
  ...

  // Read form the DP data device
  if(ReadFile(hData,(LPVOID)&Data,sizeof(Data),&nBytes,NULL))
  {
    // read DP data
    ...
  }
}
```

3.3.6 ReadFileEx

The **ReadFileEx** function reads data from a file asynchronously. It is designed solely for asynchronous operation, unlike the **ReadFile** function, which is designed for both synchronous and asynchronous operation. **ReadFileEx** lets an application perform other processing during a file read operation.
The **ReadFileEx** function reports its completion status asynchronously, calling a specified completion routine when reading is completed and the calling thread is in an alertable wait state.

```
BOOL ReadFileEx
        (
        HANDLE                              hFile,
        LPVOID                              lpBuffer,
        DWORD                               nNumberOfBytesToRead,
        LPOVERLAPPED                        lpOverlapped,
        LPOVERLAPPED_COMPLETION_ROUTINE     lpCompletionRoutine
        );
```

Function parameter description:

hFile:	An open handle that specifies the device to be read from. This file handle must have been created with the FILE_FLAG_OVERLAPPED flag and must have GENERIC_READ access to the file.
lpBuffer:	Points to a buffer that receives the data read from the file. This buffer must remain valid for the duration of the read operation. The application should not use this buffer until the read operation is completed.
nNumberOfBytesToRead:	Specifies the maximum number of bytes to be read from the file. If nNumberOfBytesToRead is zero, this function does nothing.
lpOverlapped:	Points to an **OVERLAPPED** data structure that supplies data to be used during the asynchronous (overlapped) file read operation. If the device specified by hFile supports the concept of byte offsets (these are the general data devices, e.g. "\\PROFIBUS\Board0\DpData") , the caller of **ReadFileEx** must specify a byte offset at which reading should begin. The caller specifies the byte offset by setting the **OVERLAPPED** structure's Offset member; the OffsetHigh member must be set to 0. If the file entity specified by hFile does not support the concept of byte, the caller must set the Offset and OffsetHigh members to zero, or **ReadFileEx** fails. The **ReadFileEx** function ignores the **OVERLAPPED** structure's hEvent member. An application is free to use that member for its own purposes in the context of a **ReadFileEx** call. **ReadFileEx** signals completion of its read operation by calling, or queuing a call to, the completion routine pointed to by lpCompletionRoutine, so it does not need an event handle. The **ReadFileEx** function does use the **OVERLAPPED** structure's Internal and InternalHigh members. An application should not set these members. The **OVERLAPPED** data structure pointed to by lpOverlapped must remain valid for the duration of the read operation. It should not be a variable that can go out of scope while the file read operation is in progress.
lpCompletionRoutine:	Points to the completion routine to be called when the read operation is complete and the calling thread is in an alertable wait state. For more information about the completion routine, see **FileIOCompletionRoutine**.

Possible function return values(defined in the header file PB_ERR.H)**:**

- If the function succeeds, the return value is **TRUE**.
- If the function fails, the return value is **FALSE**. To obtain extended error information, call GetLastError.

If the function succeeds, the calling thread has an asynchronous I/O operation pending: the overlapped read operation from the file. When this I/O operation completes and the calling thread is blocked in an alertable wait state, the system calls the function pointed to by lpCompletionRoutine, and the wait state completes with a return code of WAIT_IO_COMPLETION.

If the function succeeds and the file reading operation completes but the calling thread is not in an alertable wait state, the system queues the completion routine call, holding the call until the calling thread enters an alertable wait state.

NOTES:

Applications must neither read from nor write to the input buffer that a read operation is using until the read operation completes. A premature access to the input buffer may lead to corruption of the data read into that buffer.

The ReadFileEx function may fail if there are too many outstanding asynchronous I/O requests. In the event of such a failure, GetLastError can return ERROR_INVALID_USER_BUFFER or ERROR_NOT_ENOUGH_MEMORY.

If *hFile* is a handle to a named pipe or other file entity that does not support the byte-offset concept, the Offset and OffsetHigh members of the OVERLAPPED structure pointed to by *lpOverlapped* must be zero, or ReadFileEx fails.

An application uses the WaitForSingleObjectEx, WaitForMultipleObjectsEx, and SleepEx functions to enter an alertable wait state.

Usage

There is no sense in using the **ReadFileEx** function with board or data-oriented devices, because these system calls are served immediately by the PROFIBUS device drivers. A read operation may become pending only on the service-oriented devices.

Service-oriented devices:	Starts the asynchronous read of a received frame	
	lpBuffer:	Pointer to the buffer that receives the frame data
	nNumberOfBytesToRead:	The maximum size of a frame is defined in MAX_FMS_PDU_LENGTH.

3.3.7 WriteFile

The **WriteFile** function writes data to a file and is designed for both synchronous and asynchronous operation.

```
BOOL WriteFile
    (
    HANDLE          hFile,
    LPCVOID         lpBuffer,
    DWORD           nNumberOfBytesToWrite,
    LPDWORD         lpNumberOfBytesWritten,
    LPOVERLAPPED    lpOverlapped
    );
```

Function parameter description:

hFile:	Identifies the file to be written to. The file handle must have been created with GENERIC_WRITE access to the file. For asynchronous write operations, hFile can be any handle opened with the FILE_FLAG_OVERLAPPED flag by the **CreateFile** function.
lpBuffer:	Points to the buffer containing the data to be written to the file.
nNnumberOfBytesToWrite:	Specifies the number of bytes to write to the file.
lpNumberOfBytesWritten:	Points to the number of bytes written by this function call. If lpOverlapped is NULL, lpNumberOfBytesWritten cannot be NULL. If lpOverlapped is not NULL, lpNumberOfBytesWritten can be NULL. If this is an overlapped write operation, the number of bytes written can be fetched by calling **GetOverlappedResult**.
lpOverlapped:	Points to an **OVERLAPPED** structure. This structure is required if hFile was opened with FILE_FLAG_OVERLAPPED. If hFile was opened with FILE_FLAG_OVERLAPPED, the lpOverlapped parameter must not be NULL. It must point to a valid OVERLAPPED structure. If hFile was opened with FILE_FLAG_OVERLAPPED and lpOverlapped is NULL, the function can incorrectly report that the write operation is complete. If hFile was opened with FILE_FLAG_OVERLAPPED and lpOverlapped is not NULL, the write operation starts at the offset specified in the **OVERLAPPED** structure, and **WriteFile** may return before the write operation has been completed. In this case, **WriteFile** returns FALSE, and the **GetLastError** function returns ERROR_IO_PENDING. This allows the calling process to continue processing while the write operation is being completed. The event specified in the **OVERLAPPED** structure is set to the signaled state upon completion of the write operation. If hFile was not opened with FILE_FLAG_OVERLAPPED and lpOverlapped is NULL, the write operation starts at the current file position, and **WriteFile** does not return until the operation has been completed. If hFile was not opened with FILE_FLAG_OVERLAPPED and lpOverlapped is not NULL, the write operation starts at the offset specified in the **OVERLAPPED** structure, and **WriteFile** does not return until the write operation has been completed.

Possible function return values(defined in the header file PB_ERR.H)**:**

- If the function succeeds, the return value is **TRUE**.
- If the function fails, the return value is **FALSE**. To obtain extended error information, call **GetLastError**.

NOTES:

Applications must neither read from nor write to the output buffer that a write operation is using until the write operation completes. Premature access of the output buffer may lead to corruption of the data written from that buffer.

The WriteFile function may fail with ERROR_INVALID_USER_BUFFER or ERROR_NOT_ENOUGH_ MEMORY whenever there are too many outstanding asynchronous I/O requests.

Usage

Service-oriented device: Send a frame. The frame consists of the service description followed by the service and primitive-specific data.

lpBuffer:	Pointer to the frame to send
nNumberOfBytesToWrite:	Size of the frame The WriteFile function fails with the PROFIBUS error E_IF_NO_CNTRL_RES if the time-out for sending of the frame elapsed.

Data-oriented devices: Writes DP data. The DP slave data devices check the status information of the slave and return the error `E_SLAVE_ERROR` if the slave status is bad.

lpBuffer:	Pointer to the DP data which should be written.
nNumberOfBytesToWrite:	Size of the DP data to write. The maximum amount of bytes to write to DP slave data devices is the maximum write size of the slave.

Example

```
{
  HANDLE                hService;               // Handle of the general service device
  T_PROFI_SERVICE_DESCR Sdb;                                    // Service description
  T_DP_READ_REQ         DpReadReq;                                     // Read request
  BYTE                  Data[255];
  USIGN8                InvokeId
  ...

  // Open devices and create the DP data device
  ...

  // All devices open with read access

  // Send a DP_READ request

  // Fill the service description
  Sdb.layer      = DP;
  Sdb.service    = DP_READ;
  Sdb.primitive  = REQ;
  Sdb.invoke_id  = 0;
  Sdb.comm_ref   = 0;

  // Fill the DP-Read request
  ...

  // Create the frame
  memcpy(Data,&Sdb,sizeof(T_PROFI_SERVICE_DESCR));
  memcpy(Data + sizeof(T_PROFI_SERVICE_DESCR),
         &DpReadReq,sizeof(T_DP_READ_REQ));

  // Send the frame
  if(!WriteFile(hService,Data,sizeof(T_PROFI_SERVICE_DESCR) +
                sizeof(T_DP_READ_REQ),&nBytes,NULL))
  {
    // error handling
    ...
  }
}
```

3.3.8 WriteFileEx

The **WriteFileEx** function writes data to a file. It is designed solely for asynchronous operation, unlike **WriteFile**, which is designed for both synchronous and asynchronous operation. **WriteFileEx** reports its completion status asynchronously, calling a specified completion routine when writing is completed and the calling thread is in an alertable wait state.

```
BOOL WriteFileEx
        (
        HANDLE                              hFile,
        LPCVOID                             lpBuffer,
        DWORD                               nNumberOfBytesToWrite,
        LPOVERLAPPED                        lpOverlapped,
        LPOVERLAPPED_COMPLETION_ROUTINE     lpCompletionRoutine
        );
```

Function parameter description:

hFile:	An open handle that specifies the device to be written to. This file handle must have been created with the FILE_FLAG_OVERLAPPED flag and with GENERIC_WRITE access to the file.
lpBuffer:	Points to the buffer containing the data to be written to the file. This buffer must remain valid for the duration of the write operation. The caller must not use this buffer until the write operation is completed.
nNumberOfBytesToWrite:	Specifies the number of bytes to write to the file. If nNumberOfBtyesToWrite is zero, this function does nothing.
lpOverlapped:	Points to an **OVERLAPPED** data structure that supplies data to be used during the overlapped (asynchronous) write operation. For devices that support byte offsets (these are the general data devices, e.g. "\\PROFIBUS\Board0\Pb0\DpData"), you must specify a byte offset at which to start writing to the file. Specify this offset by setting the Offset member of the **OVERLAPPED** structure and setting OffsetHigh to zero. For files that do not support byte offsets, set Offset and OffsetHigh to zero, or **WriteFileEx** fails. The **WriteFileEx** function ignores the **OVERLAPPED** structure's hEvent member. An application is free to use that member for its own purposes in the context of a **WriteFileEx** call. **WriteFileEx** signals completion of its writing operation by calling, or queuing a call to, the completion routine pointed to by **lpCompletionRoutine**, so it does not need an event handle. The **WriteFileEx** function uses the Internal and InternalHigh members of the **OVERLAPPED** structure. Do not change the value of these members. The **OVERLAPPED** data structure must remain valid for the duration of the write operation. It should not be a variable that can go out of scope while the write operation is pending completion.
lpCompletionRoutine:	Points to a completion routine to be called when the write operation has been completed and the calling thread is in an alertable wait state. For more information about this completion routine, see **FileIOCompletionRoutine**.

Possible function return values(defined in the header file PB_ERR.H)**:**

- If the function succeeds, the return value is **TRUE**.
- If the function fails, the return value is **FALSE**. To obtain extended error information, call **GetLastError**.

If the WriteFileEx function succeeds, the calling thread has an asynchronous I/O operation pending: the overlapped write operation to the device. When this I/O operation finishes and the calling thread is blocked in an alertable wait state, the operating system calls the function pointed to by lpCompletionRoutine, and the wait completes with a return code of WAIT_IO_COMPLETION.

If the function succeeds and the file-writing operation finishes but the calling thread is not in an alertable wait state, the system queues the call to *lpCompletionRoutine, holding the call until the calling thread enters an alertable wait state.

NOTES:

Applications must neither read from nor write to the output buffer that a write operation is using until the write operation completes. Premature access of the output buffer may lead to corruption of the data written from that buffer.

The WriteFileEx function may fail, returning the messages ERROR_INVALID_USER_BUFFER or ERROR_NOT_ENOUGH_MEMORY if there are too many outstanding asynchronous I/O requests.

An application uses the WaitForSingleObjectEx, WaitForMultipleObjectsEx, MsgWaitForMultipleObjectsEx, SignalObjectAndWait, and SleepEx functions to enter an alertable wait state.

Usage

There is no sense in using the **WriteFileEx** function with board or data-oriented devices, because this system calls are served immediately by the PROFIBUS device drivers. Only on the service-oriented devices a write operation may become pending.

Service-oriented devices:	Starts the asynchronous send of a frame	
	lpBuffer.	Pointer to the frame to send
	nNumberOfBytesToWrite:	Size of the frame

3.3.9 GetOverlappedResult

The **GetOverlappedResult** function returns the results of an overlapped operation on the specified file, named pipe, or communications device.

```
BOOL GetOverlappedResult
        (
        HANDLE              hFile,
        LPOVERLAPPED        lpOverlapped,
        LPDWORD             lpNumberOfBytesTransferred,
        BOOL                bWait
        );
```

Function parameter description:

hFile:	Identifies the device. This is the same handle that was specified when the overlapped operation was started by a call to the ReadFile, WriteFile, or DeviceIoControl function.
lpOverlapped:	Points to an OVERLAPPED structure that was specified when the overlapped operation was started.
lpNumberOfBytesTransferred:	This value is undefined. Points to a 32-bit variable that receives the number of bytes that were actually transferred by a read or write operation.
bWait:	Specifies whether the function should wait for the pending overlapped operation to be completed. If TRUE, the function does not return until the operation has been completed. If FALSE and the operation is still pending, the function returns FALSE and the **GetLastError** function returns ERROR_IO_INCOMPLETE.

Possible function return values:

- If the function succeeds, the return value is **TRUE**.
- If the function fails, the return value is **FALSE**. To obtain extended error information, call **GetLastError**.

NOTES:

The results reported by the GetOverlappedResult function are those of the specified handle's last overlapped operation to which the specified OVERLAPPED structure was provided and for which the operation's results were pending. A pending operation is indicated when the function that started the operation returns FALSE and the GetLastError function returns ERROR_IO_PENDING. When an I/O operation is pending, the function that started the operation resets the hEvent member of the OVERLAPPED structure to the non-signaled state. Then when the pending operation has been completed, the system sets the event object to the signaled state.

If the bWait parameter is TRUE, GetOverlappedResult determines whether the pending operation has been completed by waiting for the event object to be in the signaled state.

If the hEvent member of the OVERLAPPED structure is NULL, the system uses the state of the *hFile* handle to signal when the operation has been completed. Use of file, named pipe, or communications-device handles for this purpose is discouraged. It is safer to use an event object because of the confusion that can occur when multiple simultaneous overlapped operations are performed on the same file, named pipe, or communications device. In this situation, there is no way to know which operation caused the object's state to be signaled.

Specify a manual-reset event object in the OVERLAPPED structure. If an auto-reset event object is used, the event handle must not be specified in any other wait operation in the interval between starting the overlapped operation and the call to GetOverlappedResult. For example, the event object is sometimes specified in one of the wait functions to wait for the operation's completion. When the wait function returns, the system sets an auto-reset event's state to non-signaled, and a subsequent call to GetOverlappedResult with the *bWait* parameter set to TRUE causes the function to be blocked indefinitely.

3.3.10 SetFilePointer

The only type of PROFIBUS devices that support the concept of file pointers is the general data device. The **SetFilePointer** function moves the file pointer of an open general data device.

```
DWORD SetFilePointer
        (
        HANDLE      hFile,
        LONG        lDistanceToMove,
        PLONG               lpDistanceToMoveHigh,
        DWORD       dwMoveMethod
        );
```

Function parameter description:

hFile: Identifies the file whose file pointer is to be moved. The file handle must have been created with GENERIC_READ or GENERIC_WRITE access to the file.

lDistanceToMove: Specifies the number of bytes to move the file pointer. A positive value moves the pointer forward in the file and a negative value moves it backward.

lpDistanceToMoveHigh: Must be NULL for PROFIBUS devices

dwMoveMethod: Specifies the starting point for the file pointer move. This parameter can be one of the following values:

Value	Meaning
FILE_BEGIN	The starting point is zero or the beginning of the file. If FILE_BEGIN is specified, DistanceToMove is interpreted as an unsigned location for the new file pointer.
FILE_CURRENT	The current value of the file pointer is the starting point.
FILE_END	Cannot be used for PROFIBUS devices.

Possible function return values:

- If the SetFilePointer function succeeds, the return value is the low-order doubleword of the new file pointer.
- If the function fails, the return value is 0xFFFFFFFF. To obtain extended error information, call **GetLastError.**

NOTES:

You should be careful when setting the file pointer in a multithreaded application. For example, an application whose threads share a file handle, update the file pointer, and read from the file must protect this sequence by using a critical section object or mutex object.

The PROFIBUS general data device doesnôt change the position of the file pointer with a read or write operation. The file pointer is only changed with the SetFilePointer function. You do not have to set the file pointer before every ReadFile or WriteFile call. Once set, the file pointer stays at the position.

Example

```
{
  HANDLE hData;                                    // Handle of the general data device
  DWORD  FilePointer                                                  // File pointer
  LONG   Distance;                                                 // Distance to move
  ...

  // Open DP data device
  ...

  FilePointer = SetFilePointer (hData, Distance, NULL, FILE_BEGIN)

  if (FilePointer == 0xffffffff)
  {
    // error handling
    ...
  }

  // Continue with read and write to the general data device
  ...
}
```

3.3.11 FileIOCompletionRoutine

The **FileIOCompletionRoutine** function is called when an asynchronous I/O function (**ReadFileEx** or **WriteFileEx**) is completed and the calling thread is in an alertable wait (using the **SleepEx**, **WaitForSingleObjectEx**, or **WaitForMultipleObjectsEx** function with the *fAlertable* flag set to TRUE).

```
VOID FileIOCompletionRoutine
    (
    DWORD           dwErrorCode,
    DWORD           dwNumberOfBytesTransfered,
    LPOVERLAPPED    lpOverlapped
    );
```

Function parameter description:

dwErrorCode: Specifies the I/O completion status. This parameter may be one of the following values:

Value	Meaning
0	The I/O was successful.
ERROR_HANDLE_EOF	The ReadFileEx function tried to read past the end of the file.

dwNumberOfBytesTransfered: Specifies the number of bytes transferred. If an error occurs, this parameter is zero.

lpOverlapped: Points to the **OVERLAPPED** structure specified by the asynchronous I/O function.

Windows does not use the hEvent member of the **OVERLAPPED** structure; the calling application may use this member to pass information to the completion routine. Windows does not use the **OVERLAPPED** structure after the completion routine is called, so the completion routine can de-allocate the memory used by the overlapped structure.

Possible function return values:

This function does not return a value.

NOTES:

The FileIOCompletionRoutine function is a placeholder for an application-defined or library-defined function name.

Returning from this function allows another pending I/O completion routine to be called. All waiting completion routines are called before the alertable thread's wait is satisfied with a return code of WAIT_IO_COMPLETION. Windows may call the waiting completion routines in any order. They may or may not be called in the order the I/O functions are completed.

Each time Windows calls a completion routine, it uses some of the application's stack. If the completion routine does additional asynchronous I/O and alertable waits, the stack may grow.

Usage

FileIOCompletionRoutine for ReadFileEx:

Service-oriented device: Fetches the result of the asynchronous read of a received frame.

dwNumberOfBytesTransfered:	Size of the received frame. If the size of the frame is 0, no frame was received during the time-out time

FileIOCompletionRoutine for WriteFileEx:

Service-oriented device: Fetches the result of the asynchronous write of a sent frame.

dwNumberOfBytesTransfered:	Size of the sent frame.

3.4 PROFIBUS APPLICATION PROGRAM INTERFACE (C-INTERFACE)

The PROFIBUS Application Program Interface (PAPI) provides two mechanisms for data exchange between application and protocol software and host and controller:

- a send/receive interface using request blocks for service-oriented data exchange and
- a data interface, which is used for fast cycle data exchange.

The PROFIBUS API consists of these functions:

papi_init	Initialize interface
papi_end	Shut down interface
papi_snd_req_res	Send frame
papi_rcv_con_ind	Receive frame
papi_set_data	Write data
papi_get_data	Read data
papi_set_dps_input_data	Write DP-Slave input data
papi_get_dps_input_data	Read DP-Slave input data
papi_get_dps_output_data	Read DP-Slave output data
papi_get_versions	Read version strings
papi_get_serial_device_number	Read serial device number
papi_get_last_error	Returns an additional last error code for INTERFACE-ERRORs

The service-oriented functions use the general service device of the low-level kernel mode driver and the data-oriented functions use the general DP-Master data device, DP-Slave input data device or DP-Slave output data device of the low-level kernel mode driver.

3.4.1 Data structures

The PROFIBUS Application Program Interface provides access to the service-oriented devices. A PROFIBUS frame consists of a service-independent description and a service-specific service-specific data block with parameters and data.

The data structure T_PROFI_SERVICE_DESCR describes the service to be performed by the protocol software.

Description of the service description block:

```
typedef struct T_PROFI_SERVICE_DESCR
        {
        USIGN16             comm_ref;
        USIGN8              layer;
        USIGN8              service;
        USIGN8              primitive;
        INT8                invoke_id;
        INT16               result;
        } T_PROFI_SERVICE_DESCR;
```

The service description block's elements are as follows:

- comm_ref : Communication reference ("logical channel")
- layer: Layer instance the service invocation is directed to (FMS, FMB, FM7, DP, DPS, FDLIF, FMS_USR, FMB_USR, FM7_USR, DP_USR, DPS_USR, FDLIF_USR)
- service_id : Service to be performed in the instance specified in the layer.
- primitive : Service primitives (request, indication, response, confirmation)
- invoke_id : ID to distinguish parallel service invocations
- result : Positive or negative result

The data block contains the service-specific data. Typically, for communication services these are data as described in the PROFIBUS IEC 61158-5 AND IEC 61158-6

Construction of the service-specific data blocks is described in the manuals FMS, FMB, FM7, FDLIF, DP and DP/V1.

3.4.2 Initialization and Shut down

The initialization function **papi_init** is used to initialize the PROFIBUS API and open the low-level devices of the PROFIBUS hardware driver.

3.4.2.1 Papi-Init

The **papi_init** function is used to initialize the PROFIBUS API and to open the low-level devices of the **desired interface** (board) of the PROFIBUS hardware driver. The function has to be called before any other function of PROFIBUS-API is called

```
INT16 papi_init
        (
        IN USIGN8       Board,
        IN USIGN32      ReadTimeout,
        IN USIGN32      WriteTimeout
        );
```

Function parameter description:

Board:	Desired board- / interface number
ReadTimeout:	ReceiveTimeout in msec (WAIT_FOR_EVER for infinity wait).
WriteTimeout:	Send Timeout in msec (WAIT_FOR_EVER for infinity wait).

Possible function return values(defined in the header file PB_ERR.H)**:**

- E_OK	(0x00)	Interface is initialized
- E_IF_CMI_ERROR	(0x14)	Can not set timeout values
- E_IF_SERVICE_NOT_EXECUTABLE	(0x19)	Application has called papi_init before
- E_IF_READING_REGISTRY	(0xF3)	Error reading registry
- E_IF_OS_ERROR	(0xFF)	Can not open low-level device(s)

Example

```
#include "pb_if.h"
...
{
  INT16   Result;
  USIGN8  BoardNr;
  USIGN32 ReadTimeout, WriteTimeout;
  ...
  if (E_OK == (Result = papi_init(BoardNr,ReadTimeout,WriteTimeout)))
  {
    ... // PAPI is initialized
  }
}
```

3.4.2.2 Papi-End

The **papi_end** function is used to shut down the PROFIBUS API. This means that the low-level devices will be closed.

```
INT16 papi_end
        (
        IN USIGN8       Board
        );
```

Function parameter description:

Board: Desired board- / interface number

Possible function return values(defined in the header file PB_ERR.H):

- E_OK (0x00) Shutdown excuted successfully

Example

```
...
#include "pb_if.h"
...
{
  USIGN8  BoardNr;                                          // Board (Interface) number
 // Initialize PROFIBUS API
  ...

  // Shut down PROFIBUS API
  papi_end(BoardNr);

}
```

3.4.3 Send / Receive Interface

The send/receive interface provides by means for both control flow and data flow between host and controller.

Data flow between the application and the communication is described by a service invariant and a large number of service specific data structures.

Control flow is directed by means of two functions, which control the data flow in both directions.

The two cases described above are covered by two interface functions in the Softing PROFIBUS implementations.

The **papi_snd_req_res** function is used for sending requests and responses. The **papi_rcv_con_ind** function is used to receive confirmations and indications.

3.4.3.1 Papi-Snd-Req-Res

The **papi_snd_req_res** function is used for sending requests and responses to PROFIBUS interface.

```
INT16 papi_snd_req_res
        (
        IN USIGN8                       Board,
        IN T_PROFI_SERVICE_DESCR*       pSdb,
        IN VOID*                        pData,
        );
```

Function parameter description:

Board:	Desired board- / interface number
pSdb:	Pointer to the data structure of type T_PROFI_SERVICE_DESCR
pData:	Pointer to service specific parameters and data

Possible function return values(defined in the header file PB_ERR.H)**:**

- E_OK	(0)	Function executed correctly
- E_IF_FATAL_ERROR	(7)	Unrecoverable error on PROFIBUS controller
- E_IF_NO_CNTRL_RES	(10)	Controller does not respond (CMI_TIMEOUT)
- E_IF_INVALID_LAYER	(12)	Invalid layer
- E_IF_INVALID_SERVICE	(13)	Invalid service identifier
- E_IF_INVALID_PRIMITIVE	(14)	Invalid service primitive
- E_IF_INVALID_DATA_SIZE	(15)	Not enough CMI data block memory
- E_IF_RESOURCE_UNAVAILABLE	(21)	No resource available
- E_IF_NO_PARALLEL_SERVICES	(22)	No parallel services allowed
- E_IF_SERVICE_CONSTR_CONFLICT	(23)	Service temporarily not executable
- E_IF_SERVICE_NOT_SUPPORTED	(24)	Service not supported in subset

- E_IF_SERVICE_NOT_EXECUTABLE	(25)	Service not executable
- E_IF_INVALID_PARAMETER	(30)	Invalid parameter in REQ or RES
- E_IF_PAPI_NOT_INITIALIZED	(33)	API not initialized
- E_IF_SOCKET_ERROR	(254)	OS system TCP socket error
- E_IF_OS_ERROR	(255)	OS system error

NOTES:

If papi_snd_req_res function fails with E_IF_NO_CTRL_RES, the controller did not respond during the send time-out value specified in papi_init. You can obtain extended error information with GetLastError if the function returns E_IF_OS_ERROR.

Example

```
...
#include "pb_if.h"
...
{
  T_PROFI_SERVICE_DESCR           Sdb;                      // Service description block
  T_FMB_SET_CONFIGURATION_REQ     FmbSetConfigurationReq;              // Request block
  INT16                           Result;
  USIGN8                          InvokeId;
  USIGN8                          BoardNr;                  // Board (Interface) number

  // initialize the PROFIBUS API
  ...
  // send a FMB_SET_CONFIGURATION request
  // fill the service description
  sdb.layer      = FMB;
  sdb.service    = FMB_SET_CONFIGURATION;
  sdb.primitive  = REQ;
  sdb.invoke_id  = ++invokeId;
  sdb.comm_ref   = 0;

  // fill the FMB_SET_CONFIGURATION_REQ structure
  ...

  if (E_OK != (Result = papi_snd_req_res(BoardNr,
                                         &Sdb,
                                         (void *) &FmbSetConfigurationReq,
                                        )))
  {
    // Error handling
    ...
  }
  ...
}
```

3.4.3.2 Papi-Rcv-Con-Ind

The **papi_rcv_con_ind** function is used to receive a service indication or service confirmation from the PROFIBUS interface when available.

```
INT16 papi_rcv_con_ind
        (
        IN      USIGN8                      Board,
        IN      T_PROFI_SERVICE_DESCR*      pSdb,
        IN      VOID*                       pData,
        INOUT   USIGN16*                    pDataLength
        );
```

Function parameter description:

Board:	Desired board- / interface number
pSdb:	Buffer for service description block
pData:	Buffer for service specific data block
pDataLen:	On function invocation: maximal size of data block On function return: actual size of service specific data block

The function returns **CON_IND_RECEIVED** to signal that a confirmation or indication is available.

Possible function return values (defined in the header file PB_ERR.H)**:**

- NO_CON_IND_RECEIVED	(0)	There is no confirmation or indication
- CON_IND_RECEIVED	(1)	Confirmation or indication is available
- E_IF_FATAL_ERROR	(7)	Unrecoverable error on PROFIBUS controller
- E_IF_NO_CNTRL_RES	(10)	Controller does not respond (CMI_TIMEOUT)
- E_IF_INVALID_DATA_SIZE	(15)	Size of data block provided not sufficient
- E_IF_CMI_ERROR	(20)	Serious CMI error
- E_IF_RESOURCE_UNAVAILABLE	(21)	No resource available
- E_IF_PAPI_NOT_INITIALIZED	(33)	API not initialized
- E_IF_SOCKET_ERROR	(254)	OS system TCP socket error
- E_IF_OS_ERROR	(255)	OS system error

NOTES:

Extended error information can be obtained with GetLastError if the function returns E_IF_OS_ERROR.

Example

```
...
#include "pb_if.h"
...
{
  USIGN8                 BoardNr ;                          // Board (Interface) number
  T_PROFI_SERVICE_DESCR  Sdb;                                    // Service description
  BYTE                   Data[255];                                      // Data buffer
  USIGN16                DataLen;
  INT16                  Result;

  // Initialize PROFIBUS API
  ...
  // Receive service indication or service confirmation
  for(;;)
  {
    DataLen = sizeof(Data);
    Result = papi_rcv_con_ind(BoardNr,&Sdb, &Data, &DataLen);

    if (Result == CON_IND_RECEIVED)
    {
      //  handle indication or confirmation
      ...
    }
    else
    {
      if (Result == NO_CON_IND_RECEIVED)
      {
        // nothing received
        ...
      }
      else
      {
         //Error handling
        ...
      }
    }
    ...
  }
}
```

3.4.4 Data Interface

In addition to the send/receive interface, the PROFIBUS Application Layer Interface offers a data interface which consists of data structures shared by host and controller. This data interface allows fast cyclic data transfer.

The data interface is performed by functions, which provide the data flow from and to the DPRAM area.

3.4.4.1 Papi-Set-Data

Using the **papi_set_data** function, shared data located in the DPRAM area can be written or modified.

```
INT16 papi_set_data
        (
        IN USIGN8       Board,
        IN USIGN8       DataId,
        IN USIGN16      Offset,
        IN USIGN16      DataSize,
        IN VOID*        pData,
        );
```

Function parameter description:

Board:	Desired board- / interface number
DataId:	Identifier of the specified data structure in the Data Interface
Offset:	Offset within the data structure
DataSize:	Number of bytes to be written to the DPRAM
pData:	Data block to be written

Possible values of data_id (defined in the header file PB_IF.H)**:**

ID_DP_SLAVE_IO_IMAGE	0x80	Identifier of image for slave I/O data (DP)

The structures of the data blocks are described in the service specific parts of the PROFIBUS User Manual.

Possible function return values(defined in the header file PB_ERR.H)**:**

- E_OK	(00)	Function executed correctly
- E_IF_INVALID_DATA_SIZE	(15)	Not enough CMI data block memory
- E_IF_CMI_ERROR	(20)	Serious CMI error
- E_IF_SERVICE_NOT_SUPPORTED	(24)	Identifier is not supported

- E_IF_PAPI_NOT_INITIALIZED	(33)	API not initialized
- E_IF_INVALID_DP_STATE	(242)	PROFIBUS interface is not in OPERATE state
- E_IF_SOCKET_ERROR	(254)	OS system TCP socket error
- E_IF_OS_ERROR	(255)	OS system error

NOTES:

Writing data to the ID_DP_STATUS_IMAGE is not supported in this version.

Extended error information can be obtained with GetLastError if the function returns E_IF_OS_ERROR.

Example

```
...
#include "pb_if.h"
...
{
  USIGN8  BoardNr;
  USIGN16 Offset;
  INT16   Result;

  // Initialize PROFIBUS API
...

  // Prepare and write DP data
  ...
  if (E_OK != (Result = papi_set_data(BoardNr,
                                      ID_DP_SLAVE_IO_IMAGE,
                                      Offset,
                                      sizeof(Data),
                                      &Data)))
  {
    // Error handling
    ...
  }
}
```

3.4.4.2 Papi-Get-Data

The **papi_get_data** function is used to read shared data located in the DPRAM area.

```
INT16 papi_get_data
        (
        IN      USIGN8          Board,
        IN      USIGN8          DataId,
        IN      USIGN16         Offset,
        INOUT   USIGN16*        pDataSize,
        OUT     VOID*           pData
        );
```

Function parameter description:

Board:	Desired board- / interface number
DataId:	Identifier of the specified data structure in the Data Interface
Offset:	Offset within the data structure
pDataSize:	On function invocation: maximal size of the data buffer (pData) On function return: number of bytes actually read
pData:	Pointer to data buffer

Possible values of data_id (defined in the header file PB_IF.H)**:**

ID_DP_SLAVE_IO_IMAGE	0x80	Identifier of image for slave I/O data (DP)
ID_DP_STATUS_IMAGE	0x81	Identifier of image for status data (DP)
ID_EXCEPTION_IMAGE	0xF0	Identifier of image for exception data (IF)
ID_FW_VERS_IMAGE	0xF1	Identifer of image for firmware version (IF)
ID_SERIAL_DEVICE_NUMBER	0xF2	Identifier for image for serial device number (IF)

The structures of the data blocks are described in the service specific parts (IF, DP) of the manual.

Possible function return values(defined in the header file PB_ERR.H)**:**

- E_OK	(00)	Function executed correctly
- E_IF_INVALID_DATA_SIZE	(15)	Not enough CMI data block memory
- E_IF_CMI_ERROR	(20)	Serious CMI error
- E_IF_SERVICE_NOT_SUPPORTED	(24)	Identifier is not supported
- E_IF_PAPI_NOT_INITIALIZED	(33)	API not initialized
- E_IF_INVALID_DP_STATE	(242)	PROFIBUS interface is not in OPERATE state
- E_IF_SOCKET_ERROR	(254)	OS system TCP socket error
- E_IF_OS_ERROR	(255)	OS system error

NOTES:
Extended error information can be obtained with GetLastError if the function returns E_IF_OS_ERROR.

Example

```
...
#include "pb_if.h"
...
{
  USIGN8  BoardNr;
  USIGN16 Offset;
  USIGN16 DataSize;
  INT16   Result;

  // Initialize PROFIBUS API
  ...
  // Read DP data
  DataSize = sizeof(Data);
  if (E_OK == (Result = papi_get_data(BoardNr,
                                      ID_DP_SLAVE_IO_IMAGE,
                                      Offset,
                                      &DataSize,
                                      &Data)))
  {
    // Got data from DP slave
    ...
  }
  else
  {
    // Error handling
    ...
  }
  ...
}
```

3.4.4.3 Papi-Set-Dps-Input-Data

The **papi_set_dps_input_data** function writes the input data of the DP slave to the DP-Slave input data device. It always writes the full length of the data.

```
INT16 papi_set_dps_input_data
        (
        IN USIGN8       Board,
        IN USIGN8*      pData,
        IN USIGN8       DataLength,
        OUT USIGN8*     pState
        );
```

Function parameter description:

Board:	Desired board- / interface number
pData:	Pointer to a USIGN8 variable containing the input data
DataLength:	Number of input data to be written (in bytes). If the number does not correspond with the configured length of the input data, the error message öE_IF_INVALID_DATA_SIZEô is returned.
pState:	Pointer to the current input data status with: - DPS_INPUT_STATE_FREEZE_ENABLED The slave has enabled the function for freezing the inputs. - DPS_INPUT_STATE_FREEZE_COMMAND A corresponding Global_Control command was received. Since the last time the function profi_set_dps_input_data was called the input data have been taken over as the data to be transmitted from the slave to the master.A corresponding Global_Control command for picking up the input data was received from the master. After the execution of this function the bit is reset automatically.

Possible function return values(defined in the header file PB_ERR.H)**:**

- E_OK	(00)	Function executed correctly
- E_IF_INVALID_DATA_SIZE	(15)	Too much user data
- E_IF_PAPI_NOT_INITIALIZED	(33)	API not initialized
- E_IF_SOCKET_ERROR	(254)	OS system TCP socket error
- E_IF_OS_ERROR	(255)	OS system error

NOTES:
Extended error information can be obtained with GetLastError if the function returns E_IF_OS_ERROR.

Example

```
...
#include "pb_if.h"
...
{

  USIGN8 BoardNr;                                          // Board (Interface) number
  USIGN8 DpsInputDataLength;                             // Length of DP-Slave input data
  USIGN8 DpsInputDataState;                                // DP-Slave input data state
  INT16  Result;                                                          // Return code

  // Initialize PROFIBUS API
  ...
  // Write input data and read recent input data state
  DpsInputDataLength = sizeof(DpsInputData);
  if (E_OK == (Result = papi_set_dps_input_data(BoardNr,
                                                &DpsInputData,
                                                DpsInputDataLength,
                                                &DpsInputDataState)))
  {
    // Got recent input data state
    ...
  }
  else
  {
    // Error handling
    ...
  }
  ...
}
```

3.4.4.4 Papi-Get-Dps-Input-Data

The **papi_get_dps_input_data** function reads the currently set inputs and the associated status of the DP slave from the DP-Slave input data device.

```
INT16 papi_get_dps_input_data
        (
        IN      USIGN8      Board,
        OUT     USIGN8*     pData,
        INOUT   USIGN8*     pDataLength,
        OUT     USIGN8*     pState
        );
```

Function parameter description:

Board:	Desired board- / interface number	
pData:	Pointer to a USIGN8 variable array to read the inputs of the slave.	
pDataLength:	(IN)	Pointer to a USIGN8 variable indicating the buffer size in bytes
	(OUT)	Number of input data read
pState:	Pointer to the current input data status with:	
	- DPS_INPUT_STATE_FREEZE_ENABLED The slave has enabled the function for freezing the inputs.	
	- DPS_INPUT_STATE_FREEZE_COMMAND Since the last **'papi_set_dps_input_data'** a corresponding Global_Control command has been received. The status is read-only. The bit will only be reset with the function **papi_set_dps_input_data**.	

Possible function return values(defined in the header file PB_ERR.H)**:**

- E_OK	(00)	Function executed correctly
- E_IF_INVALID_DATA_SIZE	(15)	User buffer to small
- E_IF_PAPI_NOT_INITIALIZED	(33)	API not initialized
- E_IF_SOCKET_ERROR	(254)	OS system TCP socket error
- E_IF_OS_ERROR	(255)	OS system error

NOTES:
Extended error information can be obtained with GetLastError if the function returns E_IF_OS_ERROR.

Example

```
...
#include "pb_if.h"
...
{

  USIGN8 BoardNr;                                          // Board (Interface) number
  USIGN8 DpsInputDataBufferLength;              // Length of DP-Slave input data buffer
  USIGN8 DpsInputDataState;                                 // DP-Slave input data state
  INT16  Result;                                                              // Return code

  // Initialize PROFIBUS API
  ...
  // Read input data and recent input data state
  DpsInputDataBufferLength = sizeof(DpsInputDataBuffer);
  if (E_OK == (Result = papi_get_dps_input_data(BoardNr,
                                                &DpsInputDataBuffer,
                                                &DpsInputDataBufferLength,
                                                &DpsInputDataState)))
  {
    // Input data and recent input data state
    ...
  }
  else
  {
    // Error handling
    ...
  }
  ...
}
```

3.4.4.5 Papi-Get-Dps-Output-Data

The **papi_get_dps_output_data** function reads the current outputs of the DP slave from the DP-Slave output data device.

```
INT16 papi_get_dps_output_data
        (
        IN      USIGN8      Board,
        OUT     USIGN8*     pData,
        INOUT   USIGN8*     pDataLength,
        OUT     USIGN8*     pState
        );
```

Function parameter description:

Board: Desired board- / interface number

pData: Pointer to a USIGN8 variable array to read the outputs of the slave.

pDataLength: (IN) Pointer to a USIGN8 variable indicating the buffer size in bytes
(OUT) Number of output data read

pState: Pointer to the current output data status with:

- DPS_OUTPUT_STATE_SYNC_ENABLED
  The function for freezing the outputs has been enabled.
- DPS_OUTPUT_STATE_SYNC_COMMAND
  A corresponding Global_Control command was received. Since the last time the function papi_get_dps_output_data was called, a Sync command has been received upon which received upon which new output data have been made ready. The bit is cleared automatically after access.
- DPS_OUTPUT_STATE_CLEAR_DATA
  The outputs are in failsafe state. A corresponding command was received from the master.
- DPS_OUTPUT_STATE_VALID_DATA
  No transmission errors have occurred during data transmission from the master and user data are exchanged (no timeout or watchdog error).
- DPS_OUTPUT_STATE_NEW_DATA
  New output data were received from the master. Since the last access via papi_get_dps_output_data function new data have been delivered (independent of the Sync command). With this bit you can prevent reusing old data. The bit is cleared after access.
- DPS_OUTPUT_STATE_GLOBAL_CONTROL
  Since the last time the output data were read, a Global_Control command has been received. The bit is cleared as soon as the output data have been read.

Possible function return values(defined in the header file PB_ERR.H)**:**

- E_OK	(00)	Function executed correctly
- E_IF_INVALID_DATA_SIZE	(15)	User buffer to small
- E_IF_PAPI_NOT_INITIALIZED	(33)	API not initialized
- E_IF_SOCKET_ERROR	(254)	OS system TCP socket error
- E_IF_OS_ERROR	(255)	OS system error

NOTES:
Extended error information can be obtained with GetLastError if the function returns E_IF_OS_ERROR.

Example

```
...
#include "pb_if.h"
...
{

  USIGN8 BoardNr;                                          // Board (Interface) number
  USIGN8 DpsOutputDataBufferLength;            // Length of DP-Slave output data buffer
  USIGN8 DpsOutputDataState;                              // DP-Slave output data state
  INT16  Result;                                                                 // Return code

  // Initialize PROFIBUS API
  ...

  // Read output data and current output data state
  DpsOutputDataBufferLength = sizeof(DpsOutputDataBuffer);
  if (E_OK == (Result = papi_get_dps_output_data(BoardNr,
                                                 &DpsOutputDataBuffer,
                                                 &DpsOutputDataBufferLength,
                                                 &DpsoutputDataState)))
  {
    // Output data and current output data state
    ...
  }
  else
  {
    // Error handling
    ...
  }
  ...
}
```

3.4.5 Additional Interface Functions

4.4.5.1 Papi-Get-Versions

The **papi_get_versions** function reads the version string of the PAPI dynamic link library and of the firmware on the PROFIBUS hardware.

```
INT16 papi_get_versions
        (
        IN      USIGN8      Board,
        OUT     char*       pPapiVersion,
        OUT     char*       pFirmwareVersion,
        );
```

Function parameter description:

Board:	Desired board- / interface number
pPapiVersion:	Pointer to a buffer for the version string of the PAPI DLL
pFirmwareVersion:	Pointer to a buffer for the version string of the firmware on the PROFIBUS hardware

Possible function return values (defined in the header file PB_ERR.H)**:**

- E_OK	(00)	Function executed correctly
- E_IF_NO_CNTRL_RES	(10)	Cannot open board device
- E_IF_SOCKET_ERROR	(254)	OS system TCP socket error
- E_IF_OS_ERROR	(255)	OS system error

NOTES:

Both buffers for the version strings must have at least the size of VERSION_STRING_LENGTH. The PROFIBUS API does not have to be initialized to get to get the version strings. Extended error information can be obtained with GetLastError if the function returns E_IF_OS_ERROR.

Example

```
...
#include "pb_if.h"
...
{
  USIGN8 BoardNr;                                        // Board (Interface) number
  char   PapiVersion[VERSION_STRING_LENGTH], FirmwareVersion[VERSION_STRING_LENGTH];
  INT16  Result;

  if (E_OK == (Result = papi_get_versions(BoardNr,PapiVersion,FirmwareVersion)))
  {
    // version strings
  }
}
```

3.4.5.2 Papi-Get-Serial-Device-Number

The **papi_get_serial_device_number** function reads the serial device number of the PROFIBUS hardware.

```
INT16 papi_get_serial_device_number
        (
        IN    USIGN8          Board,
        OUT USIGN32*                  pSerialDeviceNumber
        );
```

Function parameter description:

Board:	Desired board- / interface number
pSerialDeviceNumber:	Pointer to the variable for serial device number

Possible function return values (defined in the header file PB_ERR.H)**:**

- E_OK	(00)	Function executed correctly
- E_IF_NO_CNTRL_RES	(10)	Cannot open board device
- E_IF_SOCKET_ERROR	(254)	OS system TCP socket error
- E_IF_OS_ERROR	(255)	OS system error

NOTES:

The PROFIBUS API does not have to be initialized to get to get the serial device number. Extended error information can be obtained with GetLastError if the function returns E_IF_OS_ERROR.

Example

```
...
#include "pb_if.h"
...
{
  USIGN8  BoardNr;
  USIGN32 SerialDeviceNumber;
  INT16   Result;

  if (E_OK == (Result = papi_get_serial_device_number(BoardNr,&SerialDeviceNumber)))
  {
    // Serial device number
  }
}
```

4.4.5.3 Papi-Get-Last-Error

The **papi_get_last_error** function is used to return an additional error code for the interface errors.

```
INT16 papi_get_last_error
        (
        VOID
        );
```

Possible function return values (defined in the header file PB_ERR.H)**:**

0x00	No additional error code
> 0x00	Additional error code

3.4.5 INTERFACE RETURN VALUES

This chapter gives an overview of the user interface return values. All possible return values are described in the header files PB_IF.H and PB_ERR.H.

Overview of User Interface error codes and return values

Identifier	Value	Description
- E_OK	**0**	**No error occured**
- NO_CON_IND_RECEIVED	**0**	**No confirmation or indication available**
- CON_IND_RECEIVED	**1**	**Confirmation or indication ws received**
- E_IF_FATAL_ERROR	7	Unrecoverable error on board [1)]
- E_IF_INIT_INVALID_PARAMETER	8	Invalid initialization parameter
- E_IF_NO_CNTRL_RES	10	Controller does not respond
- E_IF_INVALID_CNTRL_TYPE_VERSION	11	Invalid controller type or invalid firmware version
- E_IF_INVALID_LAYER	12	Invalid layer
- E_IF_INVALID_SERVICE	13	Invalid service identifier
- E_IF_INVALID_PRIMITIVE	14	Invalid service primitive
- E_IF_INVALID_DATA_SIZE	15	Not enough CMI data block memory
- E_IF_INVALID_CMI_CALL	19	Invalid CMI call
- E_IF_CMI_ERROR	20	Error occured in CMI
- E_IF_RESOURCE_UNAVAILABLE	21	No resource available
- E_IF_NO_PARALLEL_SERVICES	22	No parallel services allowed
- E_IF_SERVICE_CONSTR_CONFLICT	23	Service temporarily not executable
- E_IF_SERVICE_NOT_SUPPORTED	24	Service not supported
- E_IF_SERVICE_NOT_EXECUTABLE	25	Service not executable
- E_IF_INVALID_ACCESS	26	Invalid access to protocol software
- E_IF_NO_CNTRL_PRESENT	28	No controller present
- E_IF_INVALID_PARAMETER	30	Invalid parameter in REQ or RES
- E_IF_INIT_FAILED	31	Init. API or Controller failed
- E_IF_EXIT_FAILED	32	Exit API or Controller failed
- E_IF_PAPI_NOT_INITIALIZED	33	API not initialized
- E_IF_NO_DEVICE_CONNECTION	34	no PROFIBUS device connection (TCP/IP)
- E_IF_SLAVE_DIAG_DATA	240	no data available
- E_IF_SLAVE_ERROR	241	no data exchange
- E_IF_INVALID_DP_STATE	242	DP is not in state clear/operate
- E_IF_READING_REGISTRY	243	Error reading registry
- E_IF_SOCKET_ERROR	254	OS system TCP Socket error
- E_IF_OS_ERROR	255	OS system (WIN,DOS) error

1) **NOTE:** If the interface error **E_IF_FATAL_ERROR** is indicated, the user can read additional information about this error via the service interface function **papi_rcv_con_ind** or data interface function **papi_get_data**:

Read additional error information via papi_rcv_con_ind:

Service-Description-Block for Indication:

USIGN16	comm_ref	0
USIGN8	layer	FMB_USR
USIGN8	service	FMB_EXCEPTION
USIGN8	primitive	IND
INT8	invoke_id	0
INT16	result	POS

Data block for Indication:

Data structure	T_EXCEPTION	
USIGN8	task_id	Task in wich the fatal system error is occurred
USIGN8	par1	Exception parameter 1
USIGN16	par2	Exception parameter 2
USIGN16	par3	Exception parameter 3

Read additional error information via papi_get_data:

```
papi_get_data (Board,                          /* board number                          */
               ID_EXCEPTION_IMAGE,             /* Identifier of the exception description */
               0,                              /* Offset in the exception description   */
               (USIGN16  FAR*)       &data_len, /* Size of the exception description     */
               (T_EXCEPTION FAR*)  &exception /* Pointer to the exception description  */
              );

T_EXCEPTION   exception;                       /* Defined in the header file PB_ERR.H */
USIGN16       data_descr_len = sizeof(T_EXCEPTION);
```

3.5 PROFIBUS APPLICATION PROGRAM INTERFACE (.NET-INTERFACE)

The PROFIBUS Application Program Interface (PAPI) provides two mechanisms for data exchange between application and protocol software and host and controller:

- a send/receive interface using request blocks for service-oriented data exchange and
- a data interface, which is used for fast cycle data exchange.

The PROFIBUS API class **CPAPI** contains the following public member functions:

Init	Initialize interface
End	Shut down interface
SendServiceRequestResponse	Send a service frame
ReceiveConfirmationIndication	Receive a service frame
SetData	Write data
GetData	Read data
SetDpsInputData	Write DP-Slave input data
GetDpsInputData	Read DP-Slave input data
GetDpsOutputData	Read DP-Slave output data
GetVersions	Read version strings
GetSerialDeviceNumber	Read serial device number
ImportBinaryDpConfigurationFile	Import DP busparameter set and DP slaveparameter sets from binary configuration file.

3.5.1 Data structures

The PROFIBUS Application Program Interface provides access to the service-oriented devices. A PROFIBUS frame consists of a service-independent description and a service-specific service-specific data block with parameters and data.

The class C_PROFI_SERVICE_DESCR describes the service to be performed by the protocol software.

Description of the service description block:

```
class C_PROFI_SERVICE_DESCR
(
    UInt16      commRef,
    byte            layer,
    byte            service,
    byte            primitive,
    char        invokeId,
    short       result
)
```

The service description block's elements are as follows:

- commRef : Communication reference ("logical channel")
- layer: Layer instance the service invocation is directed to (Fms, Fmb, Fm7, Dp, Dps, Fdlif, FmsUsr, FmbUsr, Fm7Usr, DpUsr, DpsUsr, FdlifUsr)
- service: : Service to be performed in the instance specified in the layer.
- primitive : Service primitives (request, indication, response, confirmation)
- invokeId : ID to distinguish parallel service invocations
- result : Positive or negative result

The data block contains the service-specific data. Typically, for communication services these are data as described in the PROFIBUS IEC 61158-5 AND IEC 61158-6

Construction of the service-specific data blocks is described in the manuals FMS, FMB, FM7, FDLIF, DP and DP/V1.

3.5.2 Initialization and Shut down

The initialization function **Init** is used to initialize the PROFIBUS API and open the low-level devices of the PROFIBUS hardware driver.

3.5.2.1 CPAPI-Init

The **Init** function is used to initialize the PROFIBUS API and to open the low-level devices of the **desired interface** (board) of the PROFIBUS hardware driver. The function has to be called before any other function of PROFIBUS-API is called

```
virtual void Init
        (
        IN Byte         board,
        IN UInt32       readTimeout,
        IN UInt32       writeTimeout
        );
```

Function parameter description:

board:	Desired board- / interface number
readTimeout:	ReceiveTimeout in msec (WAIT_FOR_EVER for infinity wait).
writeTimeout:	Send Timeout in msec (WAIT_FOR_EVER for infinity wait).

Possible exception values of class object CPapiException:

- NoControllerResponse	(0x0A)	Controller does not respond (CMI_TIMEOUT)
- CmiError	(0x14)	Can not set timeout values
- ServcieNotExecutable	(0x19)	Application has called Init before
- ReadingRegiatry	(0xF3)	Error reading registry
- SocketError	(0xFE)	Can not open TCP socket
- OsError	(0xFF)	Can not driver devices

Example

```
C#
...
using PapiWrapper.Common;
...
static CPapi _papi;
...
{
  Byte  boardNumber;
  UInt32 readTimeout, writeTimeout;
  ...
  try
  {
     _papi.Init(boardNumber, readTimeout, readTimeout);
  }
  catch (CPapiException papiExc)
  {
    ...
  }
   ...
}
```

3.5.2.2 CPAPI-End

The **End** function is used to shut down the PROFIBUS API. This means that the low-level devices will be closed.

virtual void End
(
IN Byte board
);

Function parameter description:

board: Desired board- / interface number

Example

```
C#
...
using PapiWrapper.Common;
...
static CPapi _papi;
...
{
  Byte  boardNumber;
  // Initialize PROFIBUS API
  ...

   ...
  // Shut down PROFIBUS API
  try
  {
     _papi.End(boardNumber);
  }
  catch (CPapiException papiExc)
  {
    ...
  }
   ...
}
```

3.5.3 Send / Receive Interface

The send/receive interface provides by means for both control flow and data flow between host and controller.

Data flow between the application and the communication is described by a service invariant and a large number of service specific data structures.

Control flow is directed by means of two functions, which control the data flow in both directions.

The two cases described above are covered by two interface functions in the Softing PROFIBUS implementations.

The **SendServiceRequestResponse** function is used for sending requests and responses. The **ReceiveServiceConfirmationIndication** function is used to receive confirmations and indications.

3.5.3.1 CPAPI-SendServiceRequestResponse

The **SendserviceRequestResponse** function is used for sending requests and responses to PROFIBUS interface.

```
virtual void SendServiceRequestResponse
        (
        IN Byte                         board,
        IN C_PROFI_SERVICE_DESCR^       profiServiceDescr,
        IN Object^                      sendData,
        );
```

Function parameter description:

board:	Desired board- / interface number
profiServiceDescr:	Reference to the data structure of type T_PROFI_SERVICE_DESCR
sendData:	Reference to service specific parameters and data

Possible exception values of class object CPapiException:

- FatalError	(7)	Unrecoverable error on PROFIBUS controller
- NoControllerResponse	(10)	Controller does not respond (CMI_TIMEOUT)
- InvalidLayer	(12)	Invalid layer
- InvalidService	(13)	Invalid service identifier
- InvalidPrimitive	(14)	Invalid service primitive
- InvalideDataSize	(15)	Not enough CMI data block memory
- ResourceUnavailable	(21)	No resource available
- NoParallelServices	(22)	No parallel services allowed
- ServiceConstraintConflict	(23)	Service temporarily not executable

- ServiceNotSupported	(24)	Service not supported in subset
- ServiceNotExecutable	(25)	Service not executable
- InvalidParameter	(30)	Invalid parameter in REQ or RES
- PapiNotInitilized	(33)	API not initialized
- SocketError	(254)	OS system TCP socket error
- OsError	(255)	OS system error

NOTES:

If „SendserviceRequestResponse" function fails with the exception „NoCntrlRes" the controller did not respond during the send time-out value specified in Init function. You can obtain extended error information with GetLastError if the function returns „OsError"

Example

```
C#
...
using PapiWrapper.Common;
...
static CPapi _papi = new CPAPI();
...
{
  // send a FMB_SET_CONFIGURATION request
  Byte                          boardNumber;                 // Board (Interface) number
  C_PROFI_SERVICE_DESCR         profiSdb        = new C_PROFI_SERVICE_DESCR();
  C_FMB_SET_CONFIGURATION_REQ setFmbConfigReq = new C_FMB_SET_CONFIGURATION_REQ();

  // prepare the service description block

  profiSdb.Layer     = (byte) LayerIdentifier.Fmb;
  profiSdb.Service   = (byte) FMBService.SetConfiguration;
  profiSdb.Primitive = (byte) ServicePrimitive.Req;

  // fill the FMB_SET_CONFIGURATION_REQ structure
  setFmbConfigReq.DpActive = (byte)Constants.PbTrue;                        // dp_active
  ...

  try
  {
    _papi.SendServiceRequestResponse(boardNumber, profiSdb, setFmbConfigReq);
  }
  catch (CPapiException papiExc)
  {
    ...
  }
}
```

3.5.3.2 CPAPI-ReceiveServiceConfirmationIndication

The **ReceiveServiceConfirmationIndication** function is used to receive a service indication or service confirmation from the PROFIBUS interface when available.

```
virtual ServiceConfirmationIndication ReceiveServiceConfirmationIndication
        (
        IN   Byte  board,
        );
```

Function parameter description:

board: Desired board- / interface number

Possible return values of class object ServiceConfirmationIndication:

The function returns with a reference to ServiceConfirmationIndication object signal that a confirmation or indication is available.

- IsIndConf	(0)	There is no confirmation or indication
	(1)	Confirmation or indication is available
- ServiceDescriptor		Service description block
- ServiceData		service specific data block

Possible exception values of class object CPapiException:

- FatalError	(7)	Unrecoverable error on PROFIBUS controller
- NoControllerResponse	(10)	Controller does not respond (CMI_TIMEOUT)
- InvalidDataSize	(15)	Size of data block provided not sufficient
- CmiError	(20)	Serious CMI error
- ResourceUnavailable	(21)	No resource available
- PapiNotInitialized	(33)	API not initialized
- SocketError	(254)	OS system TCP socket error
- OsError	(255)	OS system error

NOTES:

Extended error information can be obtained with GetLastError if the function returns ‘OsError’

Example

```
C#
...
using PapiWrapper.Common
...
static CPAPI _papi new CPAPI();
...
{
  byte  boardNumber;                                    // Board (Interface) number
  ...

  // Receive service indication or service confirmation
  try
  {
    ServiceConfirmationIndication serviceConfInd =
                                    _papi.ReceiveServiceConfirmIndication(boardNumber);
    if (serviceConfInd!= null)
    {
      if (serviceConfInd.IsIndConf) // ConIndReceived
      {
        // confirmation / indication received
        ...
      }
      else
      {
         // no confirmation and no indication received
         ...
       }
     }
  }
  catch (CPapiException papiExc)
  {
    ...
  }

...

}
```

3.5.4 Data Interface

In addition to the send/receive interface, the PROFIBUS Application Layer Interface offers a data interface which consists of data structures shared by host and controller. This data interface allows fast cyclic data transfer.

The data interface is performed by functions, which provide the data flow from and to the DPRAM area.

3.5.4.1 CPAPI-SetData

Using the **SetData** function, shared data located in the DPRAM area can be written or modified.

```
virtual void SetData
        (
        IN Byte                 board,
        IN Byte                 dataId
        IN UInt16               offset,
        IN UInt16               dataLength,
        IN array<byte>^                 data
        );
```

Function parameter description:

board:	Desired board- / interface number
dataId:	Identifier of the specified data structure in the Data Interface
offset:	Offset within the data structure
dataLengh:	Number of bytes to be written to the DPRAM
data:	Reference to data block to be written

Possible values of dataId (class DataImage)**:**

IdDPSlaveIo	0x80	Identifier of image for slave I/O data (DP)

The structures of the data blocks are described in the service specific parts of the PROFIBUS User Manual.

Possible exception values of class object CPapiException:

- InvalidDataSize	(15)	Not enough CMI data block memory
- CmiError	(20)	Serious CMI error
- ServiceNotSupported	(24)	Identifier is not supported

- PapiNotInitialized	(33)	API not initialized
- InvalidDpState	(242)	PROFIBUS interface is not in OPERATE state
- SocketError	(254)	OS system TCP socket error
- OsError	(255)	OS system error

NOTES:

Writing data to the ñDpStatusò image is not supported in this version.

Extended error information can be obtained with GetLastError if the function returns ñOsErrorò.

Example

```
C#
...
using PapiWrapper.Common
...
static CPAPI _papi new CPAPI();
...
{
  byte   boardNumber;
  ushort offset =  0;
  ushort dataSize = 6;
  byte[] data = new byte[6] { 0x2F, 0x2D, 0x5C, 0x7C, 0x37, 0x54};
  ...

  // Write DP IO data
  try
  {
    _papi.Setdata(boardNumber,
                  (byte) DataImage.IdDpSlaveIo,
                  offset,
                  dataSize,
                  data);
    ...
  }
  catch CPapiException papiExc)
  {
    ...
  }
}
```

3.5.4.2 CPAPI-GetData

The **GetData** function is used to read shared data located in the DPRAM area.

```
virtual void GetData
        (
        IN      Byte            board,
        IN      Byte            dataId,
        IN      UInt16          offset,
        INOUT   Byte%           dataLength,
        OUT     array<Byte>^    data,
        );
```

Function parameter description:

board:	Desired board- / interface number
dataId:	Identifier of the specified data structure in the Data Interface
offset:	Offset within the data structure
dataLength:	On function invocation: maximal size of the data buffer (pData) On function return: number of bytes actually read
data:	Reference to data buffer

Possible values of dataId (class DataImage,)**:**

IdDpSlaveIo	0x80	Identifier of image for slave I/O data (DP)
IdDpStatusI	0x81	Identifier of image for status data (DP)
IdException	0xF0	Identifier of image for exception data (IF)
IdFirmwareVersion	0xF1	Identifer of image for firmware version (IF)
IdSerialDeviceNumber	0xF2	Identifier for image for serial device number (IF)

The structures of the data blocks are described in the service specific parts (IF, DP) of the manual.

Possible exception values of class object CPapiException:

- InvalidDataSize	(15)	Not enough CMI data block memory
- CmiError	(20)	Serious CMI error
- ServiceNotSupported	(24)	Identifier is not supported
- PapiNotInitialized	(33)	API not initialized
- InvalidDpState	(242)	PROFIBUS interface is not in OPERATE state
- SocketError	(254)	OS system TCP socket error
- OsError	(255)	OS system error

NOTES:
Extended error information can be obtained with GetLastError if the function returns ñOsErrorò.

Example

```
C#
...
using PapiWrapper.Common
...
static CPAPI _papi new CPAPI();
...
{
  byte   boardNumber;
  ushort offset = 0;
  ushort dataSize = 244;
  byte[] data = new byte[244];
  ...
  // Read DP IO data
  try
  {
    _papi.Getdata(boardNumber,
                  (byte) DataImage.IdDpSlaveIo,
                  offset,
                  ref dataSize,
                  data);
    ...
  }
  catch CPapiException papiExc)
  {
    ...
  }
}
```

3.5.4.3 CPAPI-SetDpsInputData

The **SetDpsInputData** function writes the input data of the DP slave to the DP-Slave input data device. It always writes the full length of the data.

```
Virtual void SetDpsInputData
        (
        IN   Byte              board,
        IN   array<Byte>^      inputData,
        IN   Byte              dataLength,
        OUT  Byte%             state
        );
```

Function parameter description:

board:	Desired board- / interface number
inputData:	Reference to a byte array containing the input data
dataLength:	Number of input data to be written (in bytes). If the number does not correspond with the configured length of the input data, the exception ‘InvalidDataSize’ is returned.
state:	Reference to the current input data status with: - DPSInputState.FreezeEnabled The slave has enabled the function for freezing the inputs. - DPSInputState.FreezeCommand A corresponding Global_Control command was received. Since the last time the function **SetDpsInputData**() was called the input data have been taken over as the data to be transmitted from the slave to the master.A corresponding Global_Control command for picking up the input data was received from the master. After the execution of this function the bit is reset automatically.

Possible exception values of class object CPapiException:

- InvalidDataSize	(15)	Too much user data
- PapiNotInitialized	(33)	API not initialized
- SocketError	(254)	OS system TCP socket error
- OsError	(255)	OS system error

NOTES:
Extended error information can be obtained with GetLastError if the function returns ‘OsError’.

Example

```
C#
...
using PapiWrapper.Common
...
static CPAPI _papi new CPAPI();
...
{
  byte   boardNumber;
  byte   state;
  ushort dataSize = 6;
  byte[] data = new byte[6] { 0x2F, 0x2D, 0x5C, 0x7C, 0x37, 0x54};
  ...

  // Write DPS inputO data
  try
  {
    _papi.SetDpsInputData(boardNumber,
                          data,
                          dataSize,
                          ref state);
    ...
  }
  catch CPapiException papiExc)
  {
    ...
  }
}
```

3.5.4.4 CPAPI-GetDpsInputData

The **GetDpsInputData** function reads the currently set inputs and the associated status of the DP slave from the DP-Slave input data device.

```
Virtual void GetDpsInputData
        (
        IN      Byte            board,
        OUT     array<Byte>^    inputData,
        INOUT   Byte%           dataLength,
        OUT     Byte%           state
        );
```

Function parameter description:

board:	Desired board- / interface number	
inputData:	Reference to a byte array to read the inputs of the slave.	
dataLength:	(IN)	Referencer to a byte variable indicating the buffer size in bytes
	(OUT)	Number of input data read
state:	Reference to the current input data status with:	
	- DPSInputState.FreezeEnabled The slave has enabled the function for freezing the inputs.	
	- DPSInputState.FreezeCommand Since the last **'SetDpsInputData()'** a corresponding Global_Control command has been received. The status is read-only. The bit will only be reset with the function **SetDpsInputData()**.	

Possible exception values of class object CPapiException:

- InvalidDataSize	(15)	User buffer to small
- PapiNotInitialized	(33)	API not initialized
- SocketError	(254)	OS system TCP socket error
- OsError	(255)	OS system error

NOTES:
Extended error information can be obtained with GetLastError if the function returns ñOsErrorò.

Example

```
C#
...
using PapiWrapper.Common
...
static CPAPI _papi new CPAPI();
...
{
  byte   boardNumber;
  byte   state;
  ushort dataSize = 244;
  byte[] data = new byte[244];
  ...

  // Read DPS input data
  try
  {
    _papi.getDpsInputData(boardNumber,
                          data,
                          ref dataSize,
                          ref state);
    ...
  }
  catch CPapiException papiExc)
  {
    ...
  }
}
```

3.5.4.5 CPAPI-GetDpsOutputData

The **GetDpsOutputData** function reads the current outputs of the DP slave from the DP-Slave output data device.

```
virtual void  GetDpsOutputData
        (
        IN      Byte            board,
        OUT     array<Byte>^    outputData,
        INOUT   Byte%           dataLength,
        OUT     Byte%           state
        );
```

Function parameter description:

board: Desired board- / interface number

outputData: Reference to a byte array to read the outputs of the slave.

dataLength: (IN) Reference to a byte variable indicating the buffer size in bytes
(OUT) Number of output data read

state: Reference to the current output data status with:

- DPSOutputState.SyncEnabled
  The function for freezing the outputs has been enabled.
- DPSOutputState.SyncCommand
  A corresponding Global_Control command was received. Since the last time the function **GetDpsOutputData**() was called, a Sync command has been received upon which received upon which new output data have been made ready. The bit is cleared automatically after access.
- DPSOutputState.ClearData
  The outputs are in failsafe state. A corresponding command was received from the master.
- DPSOutputState.ValidData
  No transmission errors have occurred during data transmission from the master and user data are exchanged (no timeout or watchdog error).
- DPSOutputState.NewData
  New output data were received from the master. Since the last access via **GetDpsOutputData**() function new data have been delivered (independent of the Sync command). With this bit you can prevent reusing old data. The bit is cleared after access.
- DPSOutputState.GlobalControl
  Since the last time the output data were read, a Global_Control command has been received. The bit is cleared as soon as the output data have been read.

Possible exception values of class object CPapiException:

- InvalidDataSize	(15)	User buffer to small
- PapiNotInitialized	(33)	API not initialized
- SocketError	(254)	OS system TCP socket error
- OsError	(255)	OS system error

NOTES:
Extended error information can be obtained with GetLastError if the function returns ñOsErrorò.

Example

```
C#
...
using PapiWrapper.Common
...
static CPAPI _papi new CPAPI();
...
{
  byte   boardNumber;
  byte   state;
  ushort dataSize = 244;
  byte[] data = new byte[244];
  ...

  // Read DPS output data
  try
  {
    _papi.getDpsOutputData(boardNumber,
                           data,
                           ref dataSize,
                           ref state);
    ...
  }
  catch CPapiException papiExc)
  {
    ...
  }
}
```

3.5.5 Additional Interface Functions

3.5.5.1 CPAPI-GetVersions

The **GetVersions** function reads the version string of the PAPI dynamic link library and of the firmware on the PROFIBUS hardware.

```
virtual void GetVersions
        (
        IN      Board           board,
        OUT     String^%        papiVersion,
        OUT     String^%        firmwareVersion,
        );
```

Function parameter description:

Board:	Desired board- / interface number
papiVersion:	Reference to a buffer for the version string of the PAPI DLL
firmwareVersion:	Reference to a buffer for the version string of the firmware on the PROFIBUS hardware

Possible exception values of class object CPapiException:

- NoControllerResponse	(10)	Cannot open board device
- SocketError	(254)	OS system TCP socket error
- OsError	(255)	OS system error

NOTES:

Extended error information can be obtained with GetLastError if the function returns ñOsErrorò

Example

```
C#
...
using PapiWrapper.Common
...
static CPAPI _new CPAPI();
...
{
  byte   boardNumber;
  string papiVersion     = string.Empty;
  string firmwareVersion = string.Empty;

  try
  {
    _papi.GetVersions(boardNumber, ref papiVersion, ref firmwareVersion);
    ...
  }
  catch (CPapiException papiExc)
  {
    ...
  }
}
```

3.5.5.2 CPAPI-GetSerialDeviceNumber

The **GetSerialDeviceNumber** function reads the serial device number of the PROFIBUS hardware.

virtual UInt32 GetSerialDeviceNumber
(
IN Byte board
);

Function parameter description:

board: Desired board- / interface number

Possible function return values :

serialDeviceNumber: serial device number

Possible exception values of class object CPapiException:

- NoControllerResponse	(10)	
- SocketError	(254)	OS system TCP socket error
- OsError	(255)	OS system error

NOTES:

The PROFIBUS API does not have to be initialized to get to get the serial device number.
Extended error information can be obtained with GetLastError if the function returns ñOsErrorò.

Example

```
C#
...
using PapiWrapper.Common
...
static CPAPI _new CPAPI();
...
{
 byte boardNumber;
 try
  {
     uint serialDeviceNumber = _papi.GetSerialDeviceNumber(boardNumber);
     ...
  }
  catch (CPapiException papiExc)
  {
    ...
  }
}
```

3.5.5.3 CPAPI-ImportBinaryDpConfigurationFile

The **ImportBinaryDpConfigurationFile** function imports the the DP-Master- and DP-Slave parameter sets from a Softing DP-Configurator created configuration file.

```
virtual void ImportBinaryDpConfigurationFile
        (
        IN      String^                                     binConfigFile,
        OUT     C_DP_BUS_PARA_SET^%                         dpBusParameter,
        OUT     Dictionary<Byte, C_DP_SLAVE_PARA_SET^>^     dpSlaveParameterSets
        );
```

Function parameter description:

binConfigFile:	binary configuration file
dpBusParameter:	DP busparameter set
dpSlaveParameterSets	DP slave parameter sets

Possible exception values of class object CPapiException:

- OsError	(255)	OS system error

NOTES:

The PROFIBUS API does not have to be initialized to get to get the serial device number.
Extended error information can be obtained with GetLastError if the function returns ñOsErrorò.

3.5.6 CPAPI User Interface Exception Values

This chapter gives an overview of the user interface exception values.

Overview of User Interface Exception values

Identifier	Value	Description
- FatalError	7	Unrecoverable error on board [1)]
- InitInvalidParameter	8	Invalid initialization parameter
- NoControllerResponse	10	Controller does not respond
- InvalidCntrollertTypeVersion	11	Invalid controller type or invalid firmware version
- InvalidLayer	12	Invalid layer
- InvalidService	13	Invalid service identifier
- InvalidPrimitive	14	Invalid service primitive
- InvalidDataSize	15	Not enough CMI data block memory
- InvalidCmiCall	19	Invalid CMI call
- CmiError	20	Error occured in CMI
- ResourceUnavailable	21	No resource available
- NoParallelServices	22	No parallel services allowed
- ServiceConstraintConflict	23	Service temporarily not executable
- ServiceNotSupported	24	Service not supported
- ServiceNotExectutable	25	Service not executable
- InvalidAccess	26	Invalid access to protocol software
- NoCntrollerPresent	28	No controller present
- InvalidParameter	30	Invalid parameter in REQ or RES
- InitFailed	31	Init. API or Controller failed
- ExitFailed	32	Exit API or Controller failed
- PapiNotInitialized	33	API not initialized
- NoDeviceConnection	34	no PROFIBUS device connection (TCP/IP)
- SlaveDiagData	240	no data available
- SlaveError	241	no data exchange
- InvalidDpState	242	DP is not in state clear/operate
- ReadingRegistry	243	Error reading registry
- SocketError	254	OS system TCP Socket error
- OsError	255	OS system (WIN,DOS) error

1) **NOTE:** If the interface exception value **FatalError** is indicated, the user can read additional information about this exception via the service interface methode **ReceiveServiceConfirmationIndication() .**

Service-Description-Block for Indication:

UInt16	commRef	0
Byte	layer	LayerIdentifier.FmbUsr;
Byte	service	FMBService.Exception
Byte	primitive	ServicePrimitive.Ind
Int8	invokeId	0
Int16	result	ServiceResult.Pos

Data block for Indication:

Class C_EXCEPTION

Byte	taskId	Task in wich the fatal system error is occurred
Byte	par1	Exception parameter 1
UInt16	par2	Exception parameter 2
UInt16	par3	Exception parameter 3