# IO-Warrior Dynamic Library V1.5 for Windows

## Applicable for all IO-Warriors

**Code Mercenaries**

## Overview

The IO-Warrior Kit Dynamic Library provides a simple API to access all IO-Warrior products from Code Mercenaries. It is intended to be used with any programming language available on Windows or Linux. Sample programs are included for Microsoft VC++ 6, MS Visual Basic 6 and Borland Delphi 6 for Windows and C for Linux. The name of the library is iowkit.dll for Windows and libiowkit.so for Linux. The API is deliberately simple. It does not address plug and unplug of IO-Warriors for example.
It allows access to several IO-Warriors in parallel though. The limit is 16 IO-Warriors. If this limit is too low then it is possible to recompile the Windows DLL with a higher limit. The source code is included in the SDK.

The starting point of all activity is the function `IowKitOpenDevice()`. It opens all IO-Warriors connected to the computer. Likewise `IowKitCloseDevice()` closes all open devices. `IowKitGetNumDevs()` tells you how many devices have been found and `IowKitGetDeviceHandle()` gives access to the individual devices of the list. From there on it is mainly `IowKitRead()`, `IowKitReadNonBlockinge()` and `IowKitWrite()` to communicate with the device. The precise data to read and write is explained in the IO-Warrior data sheet "IOWarriorDatasheet.pdf".

The IO-Warriors have two communication channels. In USB terminology this is called an interface or pipe. Pipe 0 is used to directly access the I/O pins, whereas pipe 1 allows access to the special functions of the IO-Warrior. Consequently `IowKitRead()` and `IowKitWrite()` have a `numPipe` parameter. `IowKitReadImmediate()` is only for access to the I/O pins so it abstracts from the pipes and always returns 32 bits in a DWORD.

As of version 1.4 the dynmaic library is threadsafe. Also the DLL loads in Windows 95 and Windows NT, but does not find any IO-Warriors due to the lack of USB support in these Windows versions.

For Linux libiowkit.so has been implemented which exposes the same API as iowkit.dll. It needs the driver module iowkit.ko to be installed in the Linux kernel. Do not be afraid. Installing a kernel module is simple.

iowkit.dll and libiowkit.so also expose the API functions as methods for a Java class to allow Java programs to access the IO-Warrior directly. This is documented separately.

The API has been expanded to handle the new IOW56. Some examples (mainly „Simple IO“) have been expanded to handle  IOW56. The expansion is preliminary and needs some finetuning.

## Data structures

Data with the devices is exchanged in the form of reports. Each report is identified by a report ID which is the first byte of any report. For Linux the missing ReportID for Pipe0 is simulated to make the API identical to the Windows API.

The report ID assignment is per pipe. Pipe 0 for the I/O pins has only one type of report so it does not use a reportID. But Windows requires that a reportID is provided, even though the reportID is always 0. The difference between IO-Warrior24 and IO-Warrior40 is that the former has a report size of 3 bytes (report ID and 2 bytes payload), whereas the latter has a report size of 5 bytes (reportID and 4 bytes payload).It is possible to always read 5 bytes with `IowKitRead()` no matter which IO-Warrior is accessed because the read only returns in chunks of full reports and 5 bytes is too small to hold two 3 byte reports. Clean programming with correct sizes is a wiser idea though.
Pipe 1 for the special mode functions has a report size of 8 bytes for all IO-Warriors (report ID and 7 bytes payload). See "IOWarriorDatasheet.pdf" for details on allowed report IDs and the data payload for them. The header files provide some predefined structures for ease of use:

C:

```
typedef struct _IOWKIT_REPORT
 {
  UCHAR ReportID;
  union
   {
    DWORD Value;
    BYTE Bytes[4];
   };
 }
  IOWKIT_REPORT, *PIOWKIT_REPORT;

typedef struct _IOWKIT40_IO_REPORT
 {
  UCHAR ReportID;
  union
   {
    DWORD Value;
    BYTE Bytes[4];
   };
 }
  IOWKIT40_IO_REPORT, *PIOWKIT40_IO_REPORT;
```

```
typedef struct _IOWKIT24_IO_REPORT
 {
  UCHAR ReportID;
  union
   {
    WORD Value;
    BYTE Bytes[2];
  };
 }
  IOWKIT24_IO_REPORT, *PIOWKIT24_IO_REPORT;

typedef struct _IOWKIT_SPECIAL_REPORT
 {
  UCHAR ReportID;
  UCHAR Bytes[7];
 }
  IOWKIT_SPECIAL_REPORT, *PIOWKIT_SPECIAL_REPORT;
```

Delphi:

```
type
  PIOWKIT_REPORT = ^IOWKIT_REPORT;
  IOWKIT_REPORT = packed record
    ReportID: Byte;
  case Boolean of
    False: (Value: DWORD;);
    True: (Bytes: array [0..3] of Byte;);
  end;

  PIOWKIT40_IO_REPORT = ^IOWKIT40_IO_REPORT;
  IOWKIT40_IO_REPORT = packed record
    ReportID: Byte;
  case Boolean of
    False: (Value: DWORD;);
    True: (Bytes: array [0..3] of Byte;);
  end;

  PIOWKIT24_IO_REPORT = ^IOWKIT24_IO_REPORT;
  IOWKIT24_IO_REPORT = packed record
    ReportID: Byte;
  case Boolean of
    False: (Value: WORD;);
    True: (Bytes: array [0..1] of Byte;);
  end;

  PIOWKIT_SPECIAL_REPORT = ^IOWKIT_SPECIAL_REPORT;
  IOWKIT_SPECIAL_REPORT = packed record
    ReportID: Byte;
    Bytes: array [0..6] of Byte;
  end;
```

**`IOWKIT_REPORT`** dates from the 1.2 version of the API and is the same as `IOWKIT40_IO_REPORT`.

## IowKitOpenDevice

Declaration:

```
IOWKIT_HANDLE IOWKIT_API IowKitOpenDevice(void);
function IowKitOpenDevice: IOWKIT_HANDLE; stdcall;
```

Opens all available IO-Warrior devices and returns the handle to the first device found.
The value returned is an opaque handle to the specific device to be used in most of the other functions of the API.
The return value for failure is `NULL` (which is `nil` for Delphi and `0` for VB6). Use `GetLastError()` to learn more about the reason for failure. The most common reason for failure is of course that no IO-Warrior is connected. `GetLastError()` returns `ERROR_DEV_NOT_EXIST` for that.
Calling this function several times is possible, but not advisable. The devices get reenumerated and therefore the position in the list for a specific device may change.
Returning the first IO-Warrior found makes it simpler for programmers to handle the use of only one IO-Warrior.
Linux only handles a maximum of 8 IO-Warriors.

Sample usage C:

```
IOWKIT_HANDLE ioHandle;
ioHandle = IowKitOpenDevice();
if (ioHandle != NULL)
{
      // ... success, access devices
}
else
{
      // ... didn't open IoWarrior, handle error
}
```

Sample usage Delphi:

```
var
  ioHandle: IOWKIT_HANDLE;
begin
  ioHandle := IowKitOpenDevice;
  if Assigned(ioHandle) then
  begin
    // ... success, access devices
  end
  else
  begin
    // ... didn't open IoWarrior, handle error
  end;
end;
```

## IowKitSetLegacyOpenMode

**This function is only for IO-Warrior40 older than V1.0.1.0!**

Declaration:

```
BOOL IOWKIT_API IowKitSetLegacyOpenMode(ULONG legacyOpenMode);
function IowKitSetLegacyOpenMode(legacyOpenMode: ULONG): BOOL; stdcall;
```

Set legacy devices open mode.

`IowKitLegacyOpenMode()` specifies which open mode to use if old IO-Warrior40 chips are present.
`IOW_OPEN_SIMPLE` - open simple endpoints only.
`IOW_OPEN_COMPLEX` - open complex endpoints only.
Legacy open mode does not affect new IO-Warrior chips which have serial number support (Firmware V1.0.1.0 and later), IOW SDK always opens both endpoints for new IO-Warriors.
The return value is `TRUE (1)` for the parameters `IOW_OPEN_SIMPLE` and `IOW_OPEN_COMPLEX` and `FALSE (0)` otherwise.
By default IOW SDK opens only simple endpoints on old IO-Warrior 40 chips, so if you want to open the complex endpoints, you must call `IowKitSetLegacyOpenMode(IOW_OPEN_COMPLEX)` before calling `IowKitOpenDevice()`.
Note that because the SDK opens only one endpoint for each legacy device, there will always be only one pipe for each device, thus you should always use `0` as `numPipe` in calls to `IowKitRead()`/ `IowKitWrite()` functions.

As of dynamic library version 1.4 this function is deprecated. The dynamic library can now handle even old IO-Warrior40 chips without serial number fully. The only difference is now that the older firmware revisions do not implement all Special Mode functions.

Sample usage C:

```
IOWKIT_HANDLE ioHandle;

IowKitSetLegacyOpenMode(IOW_OPEN_COMPLEX);
ioHandle = IowKitOpenDevice();
```

Sample usage Delphi:

```
var
  ioHandle: IOWKIT_HANDLE;
begin
  IowKitSetLegacyOpenMode(IOW_OPEN_COMPLEX);
  ioHandle := IowKitOpenDevice;
end;
```

## IowKitGetProductId

Declaration:

```
ULONG IOWKIT_API IowKitProductId(IOWKIT_HANDLE iowHandle);
function IowKitGetProductId(devHandle: IOWKIT_HANDLE): ULONG; stdcall;
```

Return the Product ID of the IO-Warrior device identified by `iowHandle`.
The Product ID is a 16-bit Word identifying the specific kind of IO-Warrior. For easier compatibility with VB6 the function returns a 32-bit DWORD with the upper word set to 0.
`IOWKIT_PRODUCT_ID_IOW40` (`0x1500`, `$1500`, `&H1500`) is returned for an IO-Warrior 40 whereas `IOWKIT_PRODUCT_ID_IOW24` (`0x1501`, `$1501`, `&H1501`) is returned for an IO-Warrior 24. `0` is returned for an invalid `iowHandle`.
The value is cached in the dynamic library because access to the device needs some msecs.

Sample usage C:

```
BOOLEAN IsIOWarrior24(IOWKIT_HANDLE ioHandle)
{
	return IowKitGetProductId(ioHandle) == IOWKIT_PRODUCT_ID_IOW24;
}

BOOLEAN IsIOWarrior40(IOWKIT_HANDLE ioHandle)
{
	return IowKitGetProductId(ioHandle) == IOWKIT_PRODUCT_ID_IOW40;
}
```

Sample usage Delphi:

```
function IsIOWarrior24(ioHandle: IOWKIT_HANDLE): Boolean;
begin
  Result := IowKitGetProductId(ioHandle) = IOWKIT_PRODUCT_ID_IOW24;
end;

function IsIOWarrior40(ioHandle: IOWKIT_HANDLE): Boolean;
begin
  Result := IowKitGetProductId(ioHandle) = IOWKIT_PRODUCT_ID_IOW40;
end;
```

## IowKitGetNumDevs

Declaration:

```
ULONG IOWKIT_API IowKitGetNumDevs(void);
function IowKitGetNumDevs: ULONG; stdcall;
```

Returns the number of IO-Warrior devices present.
The function has to be called after `IowKitOpenDevice()` to return meaningful results.
Plugging or unplugging IO-Warriors after calling `IowKitOpenDevice()` is not handled. The number `IowKitGetNumDevs()` returns stays the same.

Sample usage C:

```
IOWKIT_HANDLE ioHandle;
ULONG numDevs;

ioHandle = IowKitOpenDevice();
if (ioHandle != NULL)
{
      // ... success, count devices
      numDevs = IowKitGetNumDevs();
}
```

Sample usage Delphi:

```
var
  ioHandle: IOWKIT_HANDLE;
  numDevs: ULONG;
begin
  ioHandle := IowKitOpenDevice;
  if Assigned(ioHandle) then
  begin
    // ... success, count devices
    numDevs := IowKitGetNumDevs;
  end;
end;
```

## IowKitGetDeviceHandle

Declaration:

```
IOWKIT_HANDLE IOWKIT_API IowKitGetDeviceHandle(ULONG numDevice);
function IowKitGetDeviceHandle(numDevice: ULONG): IOWKIT_HANDLE; stdcall;
```

Access a specific IO-Warrior present. `numDevice` is an index into the available IO-Warrior devices. The number range is `1` to `IowKitGetNumDevs()`. Any value outside that range returns `NULL/nil`. `IowKitGetDeviceHandle(1)` returns the same handle as `IowKitOpenDevice()`.
Understand this function as an extension to `IowKitOpenDevice()`. `IowKitOpenDevice()` has opened all IO-Warriors but has only returned the first one found. `IowKitGetDeviceHandle()` allows to access the other devices found.

Sample usage C:

```
IOWKIT_HANDLE ioHandles[IOWKIT_MAX_DEVICES];
ULONG numDevs, i;

ioHandles[0] = IowKitOpenDevice();
if (ioHandles[0] != NULL)
{
      // ... success, count devices
      numDevs = IowKitGetNumDevs();
      // get all IO-Warriors
      for(i = 2; i <= numDevs; i++)
      ioHandles[i-1] = IowKitGetDeviceHandle(i);
}
```

Sample usage Delphi:

```
var
  ioHandles: array [1..IOWKIT_MAX_DEVICES] of IOWKIT_HANDLE;
  I: ULONG;
begin
  ioHandles[1] := IowKitOpenDevice;
  if Assigned(ioHandles[1]) then
    // get all IO-Warriors
    for I := 2 to IowKitGetNumDevs do
      ioHandles[I] := IowKitGetDeviceHandle(I);
end;
```

## IowKitGetRevision

Declaration:

```
ULONG IOWKIT_API IowKitGetRevision(IOWKIT_HANDLE iowHandle);
function IowKitGetRevision(devHandle: IOWKIT_HANDLE): ULONG; stdcall;
```

A new function of dynamic library version 1.4.
Return the revision of the firmware of the IO-Warrior device identified by `iowHandle`.
The revision is a 16-bit Word telling the revision of the firmware. For easier compatibility with VB6 the function returns a 32-bit DWORD with the upper word set to 0.
**The revision consists of 4 hex digits.** `$1021` **designates the current revision 1.0.2.1**. `0` is returned for an invalid `iowHandle`.
Legacy IO-Warriors (without serial number) have a revision < 1.0.1.0 (`0x1010`, `$1010`**,** `&H1010`).
The value is cached in the dynamic library because access to the device needs some msecs.

Sample usage C:

```
BOOLEAN IsLegacyIOWarrior(IOWKIT_HANDLE ioHandle)
{
      return IowKitGetRevision(ioHandle) < IOW_NON_LEGACY_REVISION;
}
```

Sample usage Delphi:

```
function IsLegacyIOWarrior(ioHandle: IOWKIT_HANDLE): Boolean;
begin
  Result := IowKitGetRevision(ioHandle) < IOW_NON_LEGACY_REVISION;
end;
```

## IowKitGetSerialNumber

Declaration:

```
BOOL IOWKIT_API IowKitGetSerialNumber(IOWKIT_HANDLE iowHandle, PWCHAR serialNumber);
function IowKitGetSerialNumber(devHandle: IOWKIT_HANDLE;
  serialNumber: PWideChar): BOOL; stdcall;
```

Fills a buffer with the serial number string of the specific IO-Warrior identified by `iowHandle`.
All IO-Warriors (for IOW40 only those with firmware V1.0.1.0 and later) contain an 8 digit serial number. The serial number is represented as an Unicode string. The buffer pointed to by `serialNumber` must be big enough to hold 9 Unicode characters (18 bytes), because the string is terminated in the usual C way with a 0 character.

On success, this function copies the IO-Warrior serial number string to the buffer and returns `TRUE`. It fails and returns `FALSE` if the IO-Warrior does not have a serial number or if either `iowHandle` or `serialNumber` buffer are invalid.

Sample usage C:

```
void ShowSerialNumber(IOWKIT_HANDLE ioHandle)
{
      WCHAR buffer[9];

      IowKitGetSerialNumber(ioHandle, buffer);
      printf("%ws\n", buffer);
}
```

Sample usage Delphi:

```
procedure ShowSerialNumber(ioHandle: IOWKIT_HANDLE);
var
  Buffer: array [0..8] of WideChar;
begin
  IowKitGetSerialNumber(ioHandle, @Buffer[0]);
  ShowMessage(Buffer);
end;
```

Sample usage Visual Basic 6:

```
Dim N As Long
Dim S(18) As Byte

N = IowKitGetSerialNumber(IowKitGetDeviceHandle(1), S(0))
Label.Caption = S
```

## IowKitCloseDevice

Declaration:

```
void IOWKIT_API IowKitCloseDevice(IOWKIT_HANDLE devHandle);
procedure IowKitCloseDevice(devHandle: IOWKIT_HANDLE); stdcall;
```

Close all IO-Warriors.
You must call this function when you are done using IO-Warriors in your program.
If multiple IO-Warriors are present all will be closed by this function.

`IowKitOpenDevice()` and `IowKitCloseDevice()` use a `IOWKIT_HANDLE` for the most common case of only one IO-Warrior connected to the computer. This way you do not have to think about `IowKitGetNumDevs()` or `IowKitGetDeviceHandle()` at all.

As of dynamic library version 1.4 the function ignores the parameter completely. Since it closes all opened IO-Warriors anyway, there is no real need to check if the parameter is the `IOWKIT_HANDLE` returned by `IowKitOpenDevice()`.
The parameter is now only retained for compatibility and cleaner looking sources. If you handle only a single IO-Warrior in your program then `IowKitOpenDevice()` and `IowKitCloseDevice()` look and work as intuition suggests.

Sample usage C and Delphi:

```
        // OK, we're done, close IO-Warrior
        IowKitCloseDevice(ioHandle);
        ...
```

## IowKitRead

Declaration:

```
ULONG IOWKIT_API IowKitRead(IOWKIT_HANDLE devHandle, ULONG numPipe,
  PCHAR buffer, ULONG length);
function IowKitRead(devHandle: IOWKIT_HANDLE; numPipe: ULONG;
  buffer: PChar; length: ULONG): ULONG; stdcall;
```

Read data from IO-Warrior.
This function reads `length` bytes from IO-Warrior and returns the number of bytes read if successful.
Note that you must specify the number of the pipe (see IO-Warrior specs) to read from. `numPipe` ranges from `0` to `IOWKIT_MAX_PIPES-1`.
Since the IO-Warriors are HID devices, you can only read the data in chunks called reports. Each report is preceded by a ReportID byte. The "IOWarriorDatasheet.pdf" elaborates about that.

The function returns the number of bytes read, so you should always check if it reads the correct number of bytes, you can use `GetLastError()` to get error details. Keep in mind that data is always returned in report chunks, so reading 5 bytes from the IO-pins of an IO-Warrior 24 would only return 3 bytes of data because the IO-Warrior 24 has a 3 byte report whereas an IO-Warrior 40 has a 5 byte report.
The Special Mode pipe has a report size of 8 bytes for all IO-Warriors.
Linux does not have a ReportID byte of 0 for pipe 0 (I/O pins). To be completely compatible with Windows libiowkit.so adds that ReportID to the data.
As of dynamic library version 1.4 and later the function correctly reads several reports at once.

**ATTENTION!**
This function blocks the current thread until something changes on IO-Warrior (i.e. until user presses a button connected to an input pin, or until IIC data arrives), so if you do not want your program to be blocked you should use a separate thread for reading from IO-Warrior. If you do not want a blocking read use `IowKitReadNonBlocking()`.
Alternatively you can set the read timeout with `IowKitSetTimeout()` to force `IowKitRead()` to fail when the timeout elapsed.

Sample usage C:

```
IOWKIT40_IO_REPORT report;
ULONG res;

// Read IO pins of IO-Warrior 40
res = IowKitRead(ioHandle, IOW_PIPE_IO_PINS,
      &report, IOWKIT40_IO_REPORT_SIZE);
if (res != IOWKIT40_IO_REPORT_SIZE)
{
      // Didn't read, handle error
      ...
}
```

## IowKitReadNonBlocking

Declaration:

```
ULONG IOWKIT_API IowKitReadNonBlocking(IOWKIT_HANDLE devHandle, ULONG numPipe,
  PCHAR buffer, ULONG length);
function IowKitReadNonBlocking(devHandle: IOWKIT_HANDLE; numPipe: ULONG;
  buffer: PChar; length: ULONG): ULONG; stdcall;
```

New function of the 1.5 API.
Read data from IO-Warrior, but do not block if no data is available.
This function reads `length` bytes from IO-Warrior and returns the number of bytes read if successful.
Note that you must specify the number of the pipe (see IO-Warrior specs) to read from. `numPipe` ranges from `0` to `IOWKIT_MAX_PIPES-1`. On error or if no data is available the function returns `0`.
Since the IO-Warriors are HID devices, you can only read the data in chunks called reports. Each report is preceded by a ReportID byte. The "IOWarriorDatasheet.pdf" elaborates about that.

The function returns the number of bytes read, so you should always check if it reads the correct number of bytes, you can use `GetLastError()` to get error details. Keep in mind that data is always returned in report chunks, so reading 5 bytes from the IO-pins of an IO-Warrior 24 would only return 3 bytes of data because the IO-Warrior 24 has a 3 byte report whereas an IO-Warrior 40 has a 5 byte report.
The Special Mode pipe has a report size of 8 bytes for IO-Warriors 24 and 40.
Linux does not have a ReportID byte of 0 for pipe 0 (I/O pins). To be completely compatible with Windows libiowkit.so adds that ReportID to the data.
The function can read several reports at once. It reads as many reports as available. The internal buffering can hold up to 128 reports.

Sample usage C:

```
        IOWKIT40_IO_REPORT report;
        ULONG res;

        // Read IO pins of IO-Warrior 40
        res = IowKitReadNonBlocking(ioHandle, IOW_PIPE_IO_PINS,
              &report, IOWKIT40_IO_REPORT_SIZE);
        if (res == 0)
        {
              // Didn't read, handle error
        }
```

Sample usage Delphi:

```
        var
          Report: IOWKIT40_IO_REPORT;
          Ret: ULONG;
        begin
          // Read IO pins of IO-Warrior 40
          Ret := IowKitReadNonBlocking(ioHandle, IOW_PIPE_IO_PINS,
            @report, IOWKIT40_IO_REPORT_SIZE);
          if Ret = 0 then
          begin
            // Didn't read, handle error
          end;
```

## IowKitReadImmediate

Declaration:

```
BOOL IOWKIT_API IowKitReadImmediate(IOWKIT_HANDLE devHandle, PDWORD value);
function IowKitReadImmediate(devHandle: IOWKIT_HANDLE; var value: DWORD): BOOL; stdcall;
```

Return current value directly read from the IO-Warrior I/O pins.
The function returns `TRUE` if a new value has arrived otherwise it returns `FALSE` and places the last value read into `value`.

The function can only read the I/O pins so it does not need a `numPipe` parameter. It also abstracts from the number of I/O pins the device has. It always returns 32 bits as a DWORD. For the IOWarrior24 which has only 16 I/O pins the upper WORD of the DWORD is set to 0.

The function internally uses the Special Mode Function „Read current pin status“ if available. For chip revisions predating revision 1.0.1.0 it returns the most recent value read from the IO-Pins or returns `FALSE` if no value has been read yet.

The 1.4 version of the API  implements another strategy. That strategy is now available through `IowKitReadNonBlocking`. 1.5 reverts to the implementation of the 1.2 API to provide compatibility.

This function fails unconditionally for IOW56 devices because it cannot handle more than 32 bits. For the IOW56 you can always use the special mode function „Get Current Pin Status“ directly instead.

Sample usage C:

```
DWORD bits;

if (IowKitReadImmediate(ioHandle, &bits))
{
      // new data from IO-Warrior pins
      ...
}
```

Sample usage Delphi:

```
var
  Bits: DWORD;
begin
  if IowKitReadImmediate(ioHandle, Bits) then
  begin
    // new data from IO-Warrior pins
    ...
  end;
```

## IowKitSetTimeout

Declaration:

```
BOOL IOWKIT_API IowKitSetTimeout(IOWKIT_HANDLE devHandle, ULONG timeout);
function IowKitSetTimeout(devHandle: IOWKIT_HANDLE; timeout: ULONG): BOOL; stdcall;
```

Set read I/O timeout in milliseconds.
It is possible to lose reports with HID devices. Since reading a HID device is a blocking call it is possible to block your application in that case.

`IowKitSetTimeout()` makes `IowKitRead()` fail if it does not read a report in the allotted time.
If `IowKitRead()` times out, you have to restart any pending transaction (for example, IIC write or read transaction) from the beginning.

It is recommended to use 1 second (1000) or bigger timeout values.

Sample usage C and Delphi:

```
    // set read timeout to 1000 msecs
    IowKitSetTimeout(ioHandle, 1000);
    ...
```

## IowKitSetWriteTimeout

Declaration:

```
BOOL IOWKIT_API IowKitSetWriteTimeout(IOWKIT_HANDLE devHandle, ULONG timeout);
function IowKitSetWriteTimeout(devHandle: IOWKIT_HANDLE; timeout: ULONG): BOOL; stdcall;
```

Set write I/O timeout in milliseconds.

`IowKitSetWriteTimeout()` makes `IowKitWrite()` fail if it does not write a report in the allotted time. If `IowKitWrite()` times out, you have to restart any pending transaction (for example, IIC write transaction) from the beginning.

Failure of `IowKitWrite()` is uncommon. Check your hardware if you encounter write errors. libiowkit.so does not implement `IowKitSetWriteTimeout()` yet.

It is recommended to use 1 second (1000) or bigger timeout values.

Sample usage C and Delphi:

```
      // set write timeout to 1000 msecs
      IowKitSetWriteTimeout(ioHandle, 1000);
      ...
```

## IowKitCancelIo

Description:

```
BOOL IOWKIT_API IowKitCancelIo(IOWKIT_HANDLE devHandle, ULONG numPipe);
function IowKitCancelIo(devHandle: IOWKIT_HANDLE; numPipe: ULONG): BOOL; stdcall;
```

Cancel a read or write operation under way on one of the pipes.
This function is seldom used, because you need several threads in your program to be able to call it at all.
`IowKitRead()` blocks the thread so you need another thread for canceling. Setting the timeouts is an easier way for handling read or write problems.
The function cancels pending read and write operations simultaneously.

Sample usage:

```
      // cancel I/O for I/O pins
      IowKitCancelIo(ioHandle, IOW_PIPE_IO_PINS);
      ...
```

## IowKitWrite

Declaration:

```
ULONG IOWKIT_API IowKitWrite(IOWKIT_HANDLE devHandle, ULONG numPipe,
  PCHAR buffer, ULONG length);
function IowKitWrite(devHandle: IOWKIT_HANDLE; numPipe: ULONG;
  buffer: PChar; length: ULONG): ULONG; stdcall;
```

Write `length` bytes of data to pipe `numPipe` of IO-Warrior. The return value is the number of bytes written. Writing something else than a single report of the correct size and a valid report ID for the pipe fails for Windows. The function allows writing to the I/O pins through pipe 0 and Special Mode functions through pipe 1. To be completely compatible with the Windows version libiowkit.so expects a ReportID 0 for pipe 0 (I/O pins) even if Linux does not have a ReportID on pipe 0. The ReportID is stripped from the data sent to the device.

Sample write to pipe 0 of an IO-Warrior 40:
DWORD value consists of 32 bits, which correspond to the 32 IO-Warrior 40 I/O pins. Each bit has the following meaning:
When a 1 is written to a pin the output driver of that pin is off and the pin is pulled high by an internal resistor. The pin can now be used as an input or an output with high state.
When a 0 is written to a pin the output driver is switched on pulling the pin to ground. The pin is now a output driving low.
For example, writing 0 (all 32 bits are zero) to IO-Warrior sets all pins as outputs driving low (so if you have LEDs connected to them they will be on).
Reading the status of the pins does always return the logic level on the pins, not the value written to the pin drivers.
Writing `0xFFFFFFFF` (value in hex, all 32 bits set) sets all pins as inputs.
Note that if you want to use a pin as an input, you must first set it up as input, in other words, you must write 1 to it. For connected LEDs this means they go off.

Sample usage C:

```
IOWKIT40_IO_REPORT report;
ULONG res;

// Write IO pins of IO-Warrior 40
report.ReportID = 0;
report.Value = 0; // all LEDs *on*
res = IowKitWrite(ioHandle, IOW_PIPE_IO_PINS,
      &report, IOWKIT40_IO_REPORT_SIZE);
if (res != IOWKIT40_IO_REPORT_SIZE)
{
      // Didn't write, handle error
      ...
}
```

## IowKitGetThreadHandle

Description:

```
HANDLE IOWKIT_API IowKitGetThreadHandle(IOWKIT_HANDLE iowHandle);
function IowKitGetThreadHandle(devHandle: IOWKIT_HANDLE): THandle; stdcall;
```

A new function of dynamic library version 1.4.
It returns the internal Windows thread handle used to read the I/O pins of the IO-Warrior.
The function is only for programmers with expert knowledge about threads. It is provided for manipulations like raising or lowering thread priority.
Since Linux does not need a thread for implementing the IO-Warrior functions, `IowKitGetThreadHandle()` always returns 0 then.

Sample usage C:

```
HANDLE threadHandle;

threadHandle = IowKitGetThreadHandle(ioHandle);
if (threadHandle != NULL)
{
      // lower thread priority
      SetThreadPriority(threadHandle, THREAD_PRIORITY_BELOW_NORMAL);
}
```

Sample usage Delphi:

```
var
  ThreadHandle: Thandle;
begin
  ThreadHandle := IowKitGetThreadHandle(ioHandle);
  if ThreadHandle <> 0 then
  begin
    // lower thread priority
    SetThreadPriority(ThreadHandle, THREAD_PRIORITY_BELOW_NORMAL);
  end;
```

## IowKitVersion

Description:

```
PCHAR IOWKIT_API IowKitVersion(void);
function IowKitVersion: PChar; stdcall;
```

Return a static C string identifying the dynamic library version. This function has been added with 1.3 version of the dynamic library. Currently it returns `"IO-Warrior Kit V1.5"`.

Sample usage C:

```
    printf("%s\n", IowKitVersion());
    ...
```

Sample usage Delphi:

```
    ShowMessage(IowKitVersion);
    ...
```

Programming by Robert Marquardt.

**Legal Stuff**

Code Mercenaries
Hard- und Software GmbH
Karl-Marx-Str. 147a
12529 Schönefeld / Grossziethen
Germany
Tel: x49-3379-20509-20
Fax: x49-3379-20509-30
Mail: support@codemercs.com
Web: www.codemercs.com
HRB 9868 CB
Geschäftsführer: Guido Körber, Christian Lucht