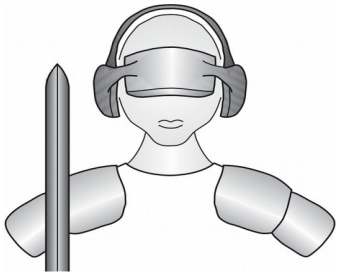
SpinWarrior Dynamic Library V1.5 for Windows and Linux

Applicable for all SpinWarriors



Code Mercenaries

Overview

The SpinWarrior Kit Dynamic Library provides a simple API to access all SpinWarrior 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 and Borland Delphi 6 for Windows and C for Linux. The name of the library is spinkit.dll for Windows and libspinkit.so for Linux.
The API is deliberately simple. It does not address plug and unplug of SpinWarriors for example.
It allows access to several SpinWarriors in parallel though. The limit is 16 SpinWarriors. If this limit is too low then it is possible to recompile the sources with a higher limit. The source code is included in the SDK.

The starting point of all activity is the function `SpinKitOpenDevice()`. It opens all SpinWarriors connected to the computer. Likewise `SpinKitCloseDevice()` closes all open devices. `SpinKitGetNumDevs()` tells you how many devices have been found and `SpinKitGetDeviceHandle()` gives access to the individual devices of the list. From there on it is mainly `SpinKitRead()` or `SpinKitReadNonBlocking()` to read from the device.

For Linux libspinkit.so has been implemented which exposes the same API as spinkit.dll.

spinkit.dll and libspinkit.so do not expose a direct Java interface yet.

The 1.5 API supports the SpinWarriors 24A3, 24R4 and 24R6.

- 24A3 supports 3 encoders using 16 bit absolute position tracking and 6 digital inputs
- 24R4 supports 4 encoders using 8 bit relative position tracking and 7 switches
- 24R6 supports 6 encoders using 8 bit relative position tracking and 3 switches

Data structures

The data from the SpinWarriors is returned in a maximum structure which can carry all data. The unused elements simply do not change. The originating SpinWarrior is included in the structure. This is for compatibility for a future 2.0 API which will use callbacks.
For a SpinWarrior 24R6 or 24R4 `Spins` elements provide a one-byte sign-extended relative value as `int`. The SpinWarrior 24A3 returns a two-byte sign-extended absolute value as `int`. `Buttons` elements provide 1 for the switch closed and 0 for the switch open.

C:

```
typedef struct _SPINKIT_DATA
 {
  SPINKIT_HANDLE Device;
  int Spins[6];
  BOOL Buttons[7];
 }
  SPINKIT_DATA, *PSPINKIT_DATA;
```

Delphi:

```
type
  PSPINKIT_DATA = ^SPINKIT_DATA;
  SPINKIT_DATA = packed record
    Device: SPINKIT_HANDLE;
    Spins: array [0..5] of Integer;
    Buttons: array [0..6] of BOOL;
  end;
```

SpinKitOpenDevice

Declaration:

```
SPINKIT_HANDLE SPINKIT_API SpinKitOpenDevice(void);
function SpinKitOpenDevice: SPINKIT_HANDLE; stdcall;
```

Opens all available SpinWarrior 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). Use `GetLastError()` to learn more about the reason for failure. The most common reason for failure is of course that no SpinWarrior is connected. `GetLastError()` returns `ERROR_DEV_NOT_EXIST` for that.
Calling this function several times is possible, but not advisable.
Returning the first SpinWarrior makes it simpler for programmers to handle the use of a single SpinWarrior.
The maximum number of devices handled is 16 for both Windows and Linux.

Sample usage C:

```
SPINKIT_HANDLE spinHandle;
spinHandle = SpinKitOpenDevice();
if (spinHandle != NULL)
{
      // ... success, access devices
}
else
{
      // ... didn't open SpinWarrior, handle error
}
```

Sample usage Delphi:

```
var
  spinHandle: SPINKIT_HANDLE;
begin
  spinHandle := SpinKitOpenDevice;
  if Assigned(spinHandle) then
  begin
    // ... success, access devices
  end
  else
  begin
    // ... didn't open SpinWarrior, handle error
  end;
end;
```

SpinKitGetProductId

Declaration:

```
ULONG SPINKIT_API SpinKitProductId(SPINKIT_HANDLE spinHandle);
function SpinKitGetProductId(spinHandle: SPINKIT_HANDLE): ULONG; stdcall;
```

Return the Product ID of the SpinWarrior device identified by `spinHandle`.
The Product ID is a 16-bit Word identifying the specific kind of SpinWarrior. For easier compatibility with VB6 the function returns a 32-bit DWORD with the upper word set to 0.
The values `SPINKIT_PRODUCT_ID24R4`, `SPINKIT_PRODUCT_ID24R6` and `SPINKIT_PRODUCT_ID24A3` can be returned. 0 is returned for an invalid `spinHandle`.
The value is cached in the dynamic library because access to the device needs some msecs.

Sample usage C:

```
BOOLEAN IsSpinWarrior24R4(SPINKIT_HANDLE spinHandle)
{
      return SpinKitGetProductId(spinHandle) == SPINKIT_PRODUCT_ID24R4;
}

BOOLEAN IsSpinWarrior24A3(SPINKIT_HANDLE spinHandle)
{
      return SpinKitGetProductId(spinHandle) == SPINKIT_PRODUCT_ID24A3;
}
```

Sample usage Delphi:

```
function IsSpinWarrior24R4(spinHandle: SPINKIT_HANDLE): Boolean;
begin
  Result := SpinKitGetProductId(spinHandle) = SPINKIT_PRODUCT_ID24R4;
end;

function IsSpinWarrior24A3(spinHandle: SPINKIT_HANDLE): Boolean;
begin
  Result := SpinKitGetProductId(spinHandle) =  SPINKIT_PRODUCT_ID24A3;
end;
```

SpinKitGetNumDevs

Declaration:

```
ULONG SPINKIT_API SpinKitGetNumDevs(void);
function SpinKitGetNumDevs: ULONG; stdcall;
```

Returns the number of SpinWarrior devices present.
The function has to be called after `SpinKitOpenDevice()` to return meaningful results.
Plugging or unplugging SpinWarriors after calling `SpinKitOpenDevice()` is not handled. The number `SpinKitGetNumDevs()` returns stays the same.

Sample usage C:

```
SPINKIT_HANDLE spinHandle;
ULONG numDevs;

spinHandle = SpinKitOpenDevice();
if (spinHandle != NULL)
{
      // ... success, count devices
      numDevs = SpinKitGetNumDevs();
}
```

Sample usage Delphi:

```
var
  spinHandle: SPINKIT_HANDLE;
  numDevs: ULONG;
begin
  spinHandle := SpinKitOpenDevice;
  if Assigned(spinHandle) then
  begin
    // ... success, count devices
     numDevs := SpinKitGetNumDevs;
  end;
end;
```

SpinKitGetDeviceHandle

Declaration:

```
SPINKIT_HANDLE SPINKIT_API SpinKitGetDeviceHandle(ULONG numDevice);
function SpinKitGetDeviceHandle(numDevice: ULONG): SPINKIT_HANDLE; stdcall;
```

Access a specific SpinWarrior. `numDevice` is an index into the available SpinWarrior devices.
The number range is `1` to `SpinKitGetNumDevs()`. Any value outside that range returns `NULL/nil`.
`SpinKitGetDeviceHandle(1)` returns the same handle as `SpinKitOpenDevice()`.
This function is an extension to `SpinKitOpenDevice()`. `SpinKitOpenDevice()` has opened all SpinWarriors but has only returned the first one found. `SpinKitGetDeviceHandle()` allows to access the remaining devices.

Sample usage C:

```
SPINKIT_HANDLE spinHandles[SPINKIT_MAX_DEVICES];
ULONG numDevs, i;

spinHandles[0] = SpinKitOpenDevice();
if (spinHandles[0] != NULL)
{
      // ... success, count devices
      numDevs = SpinKitGetNumDevs();
      // get all SpinWarriors
      for(i = 2; i <= numDevs; i++)
      spinHandles[i-1] = SpinKitGetDeviceHandle(i);
}
```

Sample usage Delphi:

```
var
  spinHandles: array [1..SPINKIT_MAX_DEVICES] of SPINKIT_HANDLE;
  I: ULONG;
begin
  spinHandles[1] := SpinKitOpenDevice;
  if Assigned(spinHandles[1]) then
    // get all SpinWarriors
    for I := 2 to SpinKitGetNumDevs do
      spinHandles[I] := SpinKitGetDeviceHandle(I);
end;
```

SpinKitGetRevision

Declaration:

```
ULONG SPINKIT_API SpinKitGetRevision(SPINKIT_HANDLE spinHandle);
function SpinKitGetRevision(spinHandle: SPINKIT_HANDLE): ULONG; stdcall;
```

Return the revision of the firmware of the SpinWarrior device identified by `spinHandle`.
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. `$1100` designates the current revision 1.1.0.0. 0 is returned for an invalid `spinHandle`. The value is cached in the dynamic library because access to the device needs some msecs.

Sample usage C:

```
BOOLEAN IsOldSpinWarrior(SPINKIT_HANDLE spinHandle)
{
      return SpinKitGetRevision(spinHandle) < 0x1100;
}
```

Sample usage Delphi:

```
function IsOldSpinWarrior(spinHandle: SPINKIT_HANDLE): Boolean;
begin
  Result := SpinKitGetRevision(spinHandle) < $1100;
end;
```

SpinKitGetSerialNumber

Declaration:

```
BOOL SPINKIT_API SpinKitGetSerialNumber(SPINKIT_HANDLE spinHandle, PWCHAR serialNumber);
function SpinKitGetSerialNumber(spinHandle: SPINKIT_HANDLE;
  serialNumber: PWideChar): BOOL; stdcall;
```

Fills a buffer with the serial number string of the specific SpinWarrior identified by `spinHandle`.
All SpinWarriors 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 SpinWarrior serial number string to the buffer and returns `TRUE`. It fails and returns `FALSE` if either `spinHandle` or `serialNumber` buffer are invalid.

Sample usage C:

```
void ShowSerialNumber(SPINKIT_HANDLE spinHandle)
{
      WCHAR buffer[9];

      SpinKitGetSerialNumber(spinHandle, buffer);
      printf("%ws\n", buffer);
}
```

Sample usage Delphi:

```
procedure ShowSerialNumber(spinHandle: SPINKIT_HANDLE);
var
  Buffer: array [0..8] of WideChar;
begin
   SpinKitGetSerialNumber(spinHandle, @Buffer[0]);
  ShowMessage(Buffer);
end;
```

SpinKitCloseDevice

Declaration:

```
void SPINKIT_API SpinKitCloseDevice(SPINKIT_HANDLE spinHandle);
procedure SpinKitCloseDevice(spinHandle: SPINKIT_HANDLE); stdcall;
```

Close all SpinWarrior devices.
You must call this function when you are done using SpinWarriors in your program.
If multiple SpinWarriors are present all will be closed by this function.

`SpinKitOpenDevice()` and `SpinKitCloseDevice()` use a `SPINKIT_HANDLE` for the case of only one SpinWarrior connected to the computer. This way you do not have to use `SpinKitGetNumDevs()` or `SpinKitGetDeviceHandle()` at all.

The function ignores the parameter completely. Since it closes all opened SpinWarriors anyway, there is no real need to check if the parameter is the `SPINKIT_HANDLE` returned by `SpinKitOpenDevice()`. The parameter is only retained for cleaner looking sources. If you handle only a single SpinWarrior in your program then `SpinKitOpenDevice()` and `SpinKitCloseDevice()` look and work as intuition suggests.

Sample usage C and Delphi:

```
// OK, we're done, close Spinarrior
SpinKitCloseDevice(spinHandle);
...
```

SpinKitRead

Declaration:

```
BOOL SPINKIT_API SpinKitRead(SPINKIT_HANDLE spinHandle, PSPINKIT_DATA SpinData);
function SpinKitRead(spinHandle: SPINKIT_HANDLE;
  var SpinData: SPINKIT_DATA): BOOL; stdcall;
```

Read data from SpinWarrior.
`TRUE` is returned for data successfully read. `FALSE` means an invalid `spinHandle` or `SpinData` buffer.

ATTENTION!
This function blocks the current thread until something changes on the SpinWarrior, so if you do not want your program to be blocked you should use a separate thread for reading from SpinWarrior. If you do not want a blocking read use `SpinKitReadNonBlocking()`.
Alternatively you can set the read timeout with `SpinKitSetTimeout()` to force `SpinKitRead()` to fail when the timeout elapsed.
The SpinWarrior 24A3 always delivers data when read so there is no blocking. In fact this device returns data each 8 msec whether it changed or not. 8 msec is the minimum time interval a low-speed HID device can be polled with.

Sample usage C:

```
SPINKIT_DATA data;

if (SpinKitRead(spinHandle, &data))
{
      // data read, interpret it
      ...
}
```

Sample usage Delphi:

```
var
  Data: SPINKIT_DATA;

if SpinKitRead(spinHandle, Data) then
begin
      // data read, interpret it
      ...
end;
```

SpinKitReadNonBlocking

Declaration:

```
BOOL SPINKIT_API SpinKitReadNonBlocking(SPINKIT_HANDLE spinHandle,
  PSPINKIT_DATA SpinData);
function SpinKitReadNonBlocking(spinHandle: SPINKIT_HANDLE;
  var SpinData: SPINKIT_DATA): BOOL; stdcall;
```

Read data from SpinWarrior, but do not block if no data is available.
The only difference to `SpinKitRead` is that the function does not block if no data is available from the device. The return value is `FALSE` for an invalid `spinHandle` or `SpinData` parameter or more commonly that no data was read. `TRUE` signals that new data was read. For a SpinWarrior 24A3 this is always the case.
Be sure to read all pending data. The SpinWarrior 24A3 is throwing data at you with the maximum data rate possible. That means new but possibly unchanged data every 8 msecs.

Sample usage C:

```
SPINKIT_DATA data;

// read until no more data available
while (SpinKitReadNonBlocking(spinHandle, &data))
{
      // interpret data
}
```

Sample usage Delphi:

```
var
  Data: SPINKIT_DATA;
  Ret: ULONG;
begin
  // read until no more data available
  while SpinKitReadNonBlocking(spinHandle, Data) do
  begin
    // interpret data
  end;
```

SpinKitSetTimeout

Declaration:

```
BOOL SPINKIT_API SpinKitSetTimeout(SPINKIT_HANDLE spinHandle, ULONG timeout);
function SpinKitSetTimeout(spinHandle: SPINKIT_HANDLE; timeout: ULONG): BOOL; stdcall;
```

Set read I/O timeout in milliseconds.
`SpinKitSetTimeout()` makes `SpinKitRead()` fail if it does not read a report in the allotted time.
It is recommended to use 1 second (1000) or bigger timeout values.

Sample usage C and Delphi:

```
// set read timeout to 1000 msecs
SpinKitSetTimeout(spinHandle, 1000);
...
```

SpinKitVersion

Description:

```
PCHAR SPINKIT_API SpinKitVersion(void);
function SpinKitVersion: PChar; stdcall;
```

Return a static C string identifying the dynamic library version. Currently it returns `"SpinWarrior Kit V1.5"`.

Sample usage C:

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

Sample usage Delphi:

```
ShowMessage(SpinKitVersion);
...
```

Extras

The Windows dynamic library spinkit.dll exports some additional functions which are not part of the SpinKit core API. These functions are helper functions to generate keyboard events. Each functions generates one or more keyboard events (through `keybd_event`). This means that the keys are generated as if coming from real keyboard entry.

```
void CMSendVirtualKeyEx(WORD Vk, BOOL KeyUp);
void CMSendScanCodeEx(WORD Scan, BOOL KeyUp);
void CMSendVirtualKey(WORD Vk);
void CMSendScanCode(WORD Scan);
void CMSendVirtualKeySequence(WORD *VkSeq, int Count);
void CMSendScanCodeSequence(WORD *ScanSeq, int Count);
void CMSendVirtualKeySequenceEx(WORD *VkSeq, BOOL *KeyUp, int Count);
void CMSendScanCodeSequenceEx(WORD *ScanSeq, BOOL *KeyUp, int Count);
void CMSendString(PCHAR Str);
char CMSendSpinChar(int Steps);
```

`CMSendVirtualKey` sends out a key press (key down and key up in succession). The parameter is a virtual key code.

`CMSendScanCode` sends out a key press (key down and key up in succession). The parameter is a scan code.

`CMSendVirtualKeyEx` sends out a key down (`KeyUp = False`) or a key up (`KeyUp = True`) event. The parameter is a virtual key code.

`CMSendScanCodeEx` sends out a key down (`KeyUp = False`) or a key up (`KeyUp = True`) event. The parameter is a scan code.

`CMSendVirtualKeySequence` sends out a sequence of key presses (key down and key up in succession). `VkSeq` points to an array of `Count` virtual key codes to send.

`CMSendScanCodeSequence` sends out a sequence of key presses (key down and key up in succession). `ScanSeq` points to an array of `Count` scan codes to send.

`CMSendVirtualKeySequenceEx` sends out a sequence of key down (`KeyUp = False`) or a key up (`KeyUp = True`) events. `VkSeq` points to an array of `Count` virtual key codes to send. `KeyUp` points to the accompanying array of Boolean values. If `KeyUp` is `NULL` then all events are handled as key down events.

`CMSendScanCodeSequenceEx` sends out a sequence of key down (`KeyUp = False`) or a key up (`KeyUp = True`) events. `ScanSeq` points to an array of `Count` scan codes to send. `KeyUp` points to the accompanying array of Boolean values. If `KeyUp` is `NULL` then all events are handled as key down events.

`CMSendString` sends out a sequence of key presses resulting in the string of ASCII chars to appear. Shift/Ctrl/Alt downs and ups are interspersed to achieve the result. `CMSendString("ABC")` results in each letter key press enclosed in Shift down and Shift up whereas `CMSendString("abc")` does not need this.

`CMSendSpinChar` implements a spin control effect with letters. The function keeps a position in a set of chars going from `' '` (space, 0x20) to `'~'`(tilde, 0x7E). `CMSendSpinChar(0)` initializes to `' '` (space, 0x20). The function sends out a single space key press. Any other value for Steps (positive or negative) advances the position in the char set with a rollover at either end. The function sends out two key presses. The first one is always '\b' (backspace) followed by the current char in the set. The return value is the current char in the set. The Delphi sample SpinWarrior shows the intended use of `CMSendSpinChar`.

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