

# The API for USB2LPT (Application Programmer's Interface)



There is a specific API to access USB2LPT. You need this API to do:

- communicate to USB2LPT not using its “dirty” redirection feature
- speed up data transfer concatenating instructions
- use all (up to 20) data lines for user I/O with individual direction and pull-up control
- upload volatile firmware snippets (plugins) to handle bit-banging by microcontroller itself (not possible for ATmega based Low-Speed adapters)
- read and write RAM, EEPROM, and flash area of the USB2LPT controller
- upload a GPIF waveform to dramatically speed up data transfer (CY7C68013A-based high-speed devices only) with no extra firmware

This API is relatively simple and uses almost no symbolic constants.

The objective while creating USB2LPT was *not* to adopt application software. In many cases, this will work well. But for some reasons, USB2LPT specific application software will have some advantages as noted above.

The strategy to communicate to USB2LPT is easy and portable accross programming languages as C, Delphi, VisualBasic, VBA, and LabVIEW. **No additional DLLs are needed.** You open communication by **CreateFile** of device “\\.\LPT1” (or “LPT2” if second device etc.), and transfer data over the **DeviceIoControl** Windows API.

## 1. Parallel port compatible access

This access enables using USB2LPT like a real parallel port (inclusive ECP and EPP if needed), without “dirty” redirection. You don't even need **InpOut32.DLL** (or similar), you don't need administrative privileges, and you won't open a security hole. Note that **InpOut32.DLL** opens a security hole by itself; application code may format your hard disk.

### 1.1. Opening of USB2LPT

A global variable **hAccess** holding the handle to USB2LPT device is handy. ☐ Show line numbers ☐ Wrap lines

```
HANDLE hAccess;

for (int n=0; n; n--) {           // try backwards
    TCHAR DevName[12];
    wsprintf(DevName, "\\\\.\\LPT%u", n);
    hAccess = CreateFile(DevName, GENERIC_READ|GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0, 0);
    if (hAccess!=INVALID_HANDLE_VALUE) goto found;
}
hAccess = 0;
found:
```

You can open regular parallel ports with same system call. Because that's not intended yet, a USB2LPT check should follow.

```
// Read the 8051 XRAM or ATmega flash address 6 where you get the firmware date as FAT date.
// If this fails, this is not a USB2LPT.
#include "usb2lpt.h"
WORD addr = 6;
WORD date = 0;
DWORD BytesRet;
if (DeviceIoControl(hAccess, IOCTL_VLPT_XramRead/*0x22228E*/, &adr, sizeof(adr), &date, sizeof(date), &BytesRet, NULL)) {
    // this is a USB2LPT device of any version, and the firmware date can be made visible using
    FILETIME ft;
    DosDateTimeToFileTime(date, 0, &ft);
    SYSTEMTIME st;
    FileTimeToSystemTime(&ft, &st);
    TCHAR s[20]; // the string buffer where the date is put
    GetDateFormat(LOCALE_USER_DEFAULT, 0, &st, NULL, s, 20);
    // ...
}else{
    // this is a standard parallel port or something else
}
```




An “upper filter driver” for regular parallel ports is indented to be written that is compatible with USB2LPT. So an application software may use the same security-hole-free API for parallel ports later.

### 1.2. A single OUT access

This function looks like follows:

```
void outb(BYTE a, BYTE b) {
    BYTE IoData[2];
    DWORD BytesRet;
    IoData[0] = a;
    IoData[1] = b;
    DeviceIoControl(hAccess, IOCTL_VLPT_OutIn/*0x222010*/, IoData, sizeof(IoData), NULL, 0, &BytesRet, NULL);
}
```

Where is: **a** a relative address byte:

- 0 = data port (see  [Beyond Logic](#))
- 2 = control port (with bits like “Strobe”, “AutoFeed”, but “Direction” (do data port) too)
- 3 = EPP address write cycle (see  [Beyond Logic](#))
- 4 = EPP data write cycle (**a** = 5, 6 and 7 do data cycles too)
- 8 = ECP FIFO write (see  [Beyond Logic](#))
- 10 = set ECP configuration register “ECR”

Note that three bits at **a**=2 are inverted! You can avoid inversion by a “feature register”, see later. Moreover, you may output data to status port (**a**=1), but without precautions (later), this data will never appear somewhere.


You may wondering that a DLL call is much simpler. But beware! For opening a DLL, you most often need extra coding with **LoadLibrary** and **GetProcAddress**!

### 1.3. A single IN access


This function looks like follows:

```
BYTE inb(BYTE a) {
    BYTE IoData[1];
    DWORD BytesRet;
    IoData[0] = a|0x10; // set the bit for read operations
    DeviceIoControl(hAccess, IOCTL_VLPT_OutIn/*0x222010*/, IoData, sizeof(IoData), IoData, sizeof(IoData), &BytesRet, NULL);
    return IoData[0];
}
```

Same as for OUT, **a** is a relative address. The operation reads the levels at port pins (not necessarily the same as the data output). The results are:

- 0 = data port
- 1 = status port (with bits “Busy”, “Acknowledge”, “Paper End” etc.)
- 2 = control port (real line states)
- 3 = EPP address read cycle (see  [Beyond Logic](#))
- 4 = EPP data read cycle (**a** = 5, 6 and 7 do data cycles too)
- 8 = ECP FIFO read or reading of “Configuration Register A”
- 9 = read “Configuration Register B” (always 0)
- 10 = read ECP Configuration Register “ECR” (e.g. FIFO state)

Note that one bit at status port (**a**=1) and three bits at control port (**a**=2) are inverted! You can avoid inversion by an “extra” register, see later.

See this [a bit too simple wrapper DLL implementation](#)  as a reference.

### 1.4. Combined access


Singe accesses have no improvement for speed compared to port access trapping! The opposite may the case, OUT accesses are *not* cached and concatenated automatically.

When you look at the code snippets, you will see that all is done with *one* IOCTL codes (0x222010) and up to two buffers. You can concatenate IN and OUT instructions in any order upto buffer sizes of 64 Bytes (more is not tested yet, but upto 4096 bytes should work).

You combine the accesses you need by concatenating OUT addresses, following OUT data, and IN addresses into the IOCTL *input* puffer (that is, the data that goes OUT to the USB device), and reserve one byte per IN access in the IOCTL output buffer (that is, the data that comes IN from the USB device). That's all! In this way, you generate microcode that is executed by USB2LPT's firmware.

For example:

```
// Read all 17 port pin states with one DeviceIoControl invocation
void GetPinStates(BYTE states[3]) {
    static const BYTE SendBytes[3] = {0x10, 0x11, 0x12};
    DWORD BytesRet;
    DeviceIoControl(hAccess, IOCTL_VLPT_OutIn/*0x222010*/, SendBytes, 3, states, 3, &BytesRet, NULL);
}
```

Bear in mind that reading from data port does *not* read the pin states if the port is in ECP mode. Instead, the  [FIFO](#) is read.

## 2. Extended port access

A USB2LPT device ist a Multi-I/O device with 17 digital I/Os; newer revisions have even 20 useable I/Os. Using the extended port access you can set the data, data direction, and the pull-up (applies to low-speed) for every single pin.

The “backdoor” for doing this are simply some more addresses:

- 12 = direction register for data port (default: 0xFF, i.e. all outputs)
- 13 = direction register for status port (default: 0x07)
- 14 = direction register for control port (default: 0x0F)
- 15 = USB2LPT Feature Register (default: 0x00, High-Speed: 0x24)

Direction bits = 1 are outputs, otherwise inputs.

- For Cypress controller based Full-Speed and High-Speed USB2LPT, inputs always have a pull-up, either strong to 5 V (external) or weak to 3.3 V (internal).
- For ATmega based Low-Speed USB2LPT, a weak pull-up is active for inputs when corresponding output data bits are ones.

The three “lower” bits of status ports are ground pins at SubD pins 21, 22, and 23 (see schematic).

The address 11 is currently not used.

The bits of the USB2LPT Feature Register:

- Simulation of open-collector for data port
- Simulation of open-collector for control port in SPP mode
- Simulation of open-collector for control port in other modes
- unused*
- Serialized: The USB device enumerates with its serial number. While Windows normally “track” devices using the topology (of USB ports and hubs) used, the system will now change to “track” a particular device by its unique serial number. Of-course it will be in effect after next USB enumeration, i.e. device detection process.
- DarkBlue: darker blue LED using 12.5 % pulse width modulation (High-Speed only)
- DirectIO (no inversions; access to three extra I/Os located at bits 0..2 of status port, no automated open-collector simulation)
  - PullUps: Global deactivation of pull-up resistors (High-Speed only), 3.3V compatibility
  - UseBulk: Use bulk transfers instead of Interrupt Transfer (Low-Speed only). Setting this bit will make USB2LPT roughly 8 times faster, but renders this device unuseable on Windows Vista and newer. Therefore, setting this bit is generally not recommended.

The simulation of open-collector enables collision-free wired-AND logic. That is needed for some hardware.

*High-Speed, Full-Speed only:* Open-collector simulation makes the port compatible to some hardware that requires 5 V HIGH levels. Otherwise, output HIGH level is limited to 3.3 V.

*Low-Speed only:* The simulation of open-collector should only be used with external pull-up resistors, as internal pullups are very weak (about 40 kΩ). The regular output HIGH level is 5 V. If you need 3.3 V outputs, you should modify the USB2LPT device by inserting a voltage-dropping dual diode D2 like BAV199 (cut SJJ3), and change the USB pullup resistor R1 to 1.5 kΩ. See [schematic](#).

## 3. Wait Cycles and Bit Access

Combination of many OUT and IN accesses into one packet may require some sort of program controlling. There are some additional opcodes for doing this.

- 0x20 = Wait, then wait time in steps of 4 μs, 0 = 1024 μs
- 0x21 = Set single bit to 1, bit address\* follows
- 0x22 = Reset single bit to 0, bit address\* follows
- 0x23 = Toggle bit, bit address\* follows
- 0x24 = Wait for bit, bit address\* follows:
  - Bit 7:4 = Port offset, 0 = Data port, 1 = Status port, 2 = Control port
  - Bit 3 (for opcode 0x24) = Expected Bit value: 0 = continue when 0, block while 1; 1 = continue when 1, block while 0
  - Bit 3 (for opcodes 0x21, 0x22, 0x23) = Stuck-safety bit: 0 = normal LPT emulation (read from input port, may influence other bits); 1 = stuck-safe operation (read from output port, never influence other bits)
  - Bit 2:0 = bit number 0..7 (for control port, 4..7 may behave undefined)

These extra addresses behave like OUT Addresses (bit 4 cleared), i.e. a data byte follows, and there is no answer.

Conditional abort and Loop are not implemented yet, and behaviour may change. Nesting may not be allowed.

*High-Speed only:*

## 4. Planned: GPIF boosted High-Speed (48 MByte/s) transfers

High-Speed transfers are such using the GPIF (General Purpose Interface). Data rates in maximum speed will become possible, even via the 8-bit bus. This is 48 MByte/s. Fast enough for filling large RAM-based FPGAs (Spartan-3) with configuration data in a fraction of a second, and also for streaming video data from/to that FPGA.

The desired waveform must be loaded into a specific RAM area. The needed GPIF outputs (CLKOUT, IFCLK, CTL0..2) and inputs (STAT0..2, maybe IFCLK) are routed to control vs. status ports. See schematic.

While switched to High-Speed transfer, the normal parallel-port emulation is disabled.

The exact specification will be made later.

## 5. Not planned: Use of peripheral features (Timer, ADC etc.)


Such use is by far much more than what USB2LPT is planned for, and is not compatible between Low-Speed, Full-Speed and High-Speed. If someone need such, one can extend the firmware. See below.

## 6. Read and write RAM, EEPROM, or Flash

*Fully implemented. See “brenner.c”.*

For Full-Speed and High-Speed devices, the API can access:

- [bRequest=0xA0]** Restricted 8051 XRAM area (write-only)
- [bRequest=0xA2, wIndex==0]** Entire boot EEPROM location (read and write)
- [bRequest=0xA2, wIndex!=0]** Any optional I<sup>2</sup>C device on I<sup>2</sup>C bus with automated 1-byte and 2-byte address generation and auto-packetizing in any 2<sup>n</sup> quantity.
  - wIndexL = I<sup>2</sup>C address (bit 0 is not used, should be zero)
  - wIndexH = bit mask
    - Bit 1:0 = address width selector:
      - Do not prepend an address
      - Auto-prepend single-byte address out of wValueL
      - Auto-prepend double-byte address out of wValue
      - Auto-prepend double-byte address out of wValue but swap the bytes
    - Bit 2 = 1: Do not generate I<sup>2</sup>C start condition
    - Bit 3 = 1: Do not generate I<sup>2</sup>C stop condition
    - Bit 7:4 = automatic block-write selector:
      - Automatic detection: 8 for 1 byte address size, 32 for 2 byte address size
      - Block size = 1, i.e. every byte is prepended by extra (auto-incremented) address byte(s)
      - Block size = 2
      - Block size = 4
      - Block size = 8, etc.
- [bRequest=0xA3, wIndex==0]** Entire 8051 XRAM area (read and write), inclusive program memory, special-function register, USB I/O buffer area, and GPIF table location
- [bRequest=0xA3, wIndex!=0]** RAM/IRAM area (read and write, TODO)

This API allows both self-programming the EEPROM (i.e. firmware update or deletion) but also temporary firmware changes (by writing to the microcontroller's RAM). The latter option enables using the USB2LPT device as a fully customizable microcontroller device. Either you can use the GPIF interface, or do some  [bit-banging](#) in firmware, or both. The CY7C68013A doesn't expose timers or serial interfaces to its general-purpose I/O pins, so bit-banging is always necessary to implement some sort of serial protocol like SPI, I<sup>2</sup>C, or JTAG. However, bit-banging in a microcontroller using assembly language can be still quite fast.

For Low-Speed (V-USB based) devices, the API can access:

- [bRequest=0xA2]** EEPROM memory (read-write)
- [bRequest=0xA3, wIndex==0]** Flash memory (read-only)
- [bRequest=0xA3, wIndex!=0]** RAM and mapped registers (read-write)

Self-programming via this API is not possible, but you can start the bootloader by writing a magic byte 0x42 into EEPROM address 0, then generate a USB RESET, then access via  [bootloadHID](#) protocol.