Nowind USB Communication Specification
Version 4.0

By Jan Wilmans and Aaldert Dekker

Index

Introduction

The Nowind Interface connects any MSX computer to an USB host computer. The main purpose of the interface is to allow the MSX access to modern media. This media includes but is not limited to: floppy disks, harddisks, CD/DVD, flashcards and memory sticks. In the most basic mode of operation the host offers an image (file) of a 720kB MSX floppy disk to the MSX that acts in the same way a normal floppy would when inserted physically into the MSX. More advanced behaviour includes access to network resources, internet and USB devices like keyboards, mice, etc. Basically anything connected to host can be made accessible to the MSX, but the host computer is always needed as an intermediate.

USB Communication Protocol

The MSX and host computer communicate using a client/server protocol, where the MSX acts as a client and the host provides services. The host is (normally) passive and waits for requests / commands from the client. However, there are also ways for the host to send requests to the MSX. The different modes of operation are described in this chapter along with the commands available in each mode.

Service Mode

This is the default/startup mode of operation. The following commands/requests can be sent from the MSX to the host.

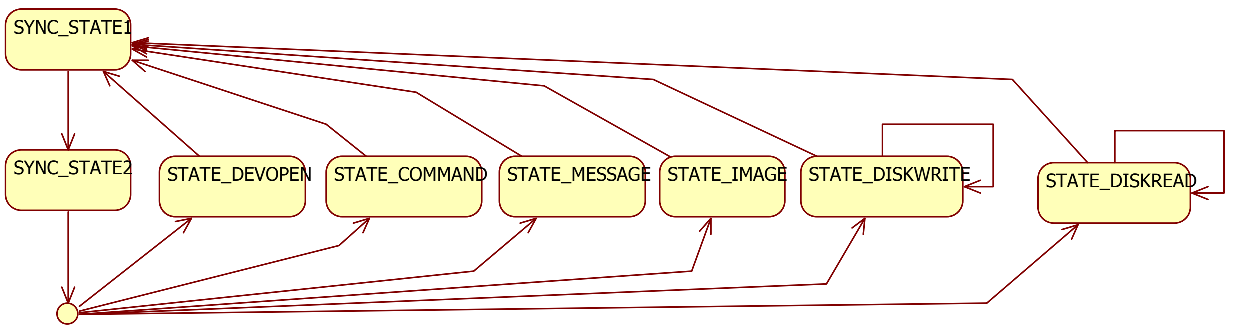


Figure 1 Service-Mode State Machine

Commands from the MSX to the host are sent in a fixed 11-byte format:

AF	05	C	B	E	D	L	H	F	A	CMD

All commands sent to the host are preceded with two bytes '0xAF 0x05', this header is used to synchronize the data-stream; if the communication is interrupted any left-over data in the receive buffers (at the host) will be purged (read and ignored) until a new command is received.

Theatrically there exists a change the left-over data contains AF 05, this is seen as an acceptable risk. Note that this risk can be easily further reduced by introducing more 'magic numbers' after the 05. So far we have not experienced any 'rogue' commands and practically it is very unlikely to happen as the MSX will time-out after 0.7 seconds of communication silence and stop sending data.
Also the host will flush send and receive buffer if a host time-out occurs (timeout value?)

After AF 05 several Z80 registers are send the order is mostly arbitrarily chosen, but consistently in little-endian byte order (little end first, meaning first the low byte, then the high byte). The accumulator and flags (AF) are sent last, because this has coding advantages.

Commands

CMD 0x0D-0x30 are BDOS commands
CMD 0x80-0x86 are DISKROM command
CMD 0x87-0x93 are commands for support of special features.

```
0x0D: BDOS_0DH_DiskReset()
0x0F: BDOS_0FH_OpenFile()
0x10: BDOS_10H_CloseFile()
0x11: BDOS_11H_FindFirst()
0x12: BDOS_12H_FindNext()
0x13: BDOS_13H_DeleteFile()
0x14: BDOS_14H_ReadSeq()
0x15: BDOS_15H_WriteSeq()
0x16: BDOS_16H_CreateFile()
0x17: BDOS_17H_RenameFile()
0x21: BDOS_21H_ReadRandomFile()
0x22: BDOS_22H_WriteRandomFile()
0x23: BDOS_23H_GetFileSize()
0x24: BDOS_24H_SetRandomRecordField()
0x26: BDOS_26H_WriteRandomBlock()
0x27: BDOS_27H_ReadRandomBlock()
0x28: BDOS_28H_WriteRandomFileWithZeros()
0x2A: BDOS_2AH_GetDate()
0x2B: BDOS_2BH_SetDate()
0x2C: BDOS_2CH_GetTime()
0x2D: BDOS_2DH_SetTime()
0x2E: BDOS_2EH_Verify()
0x2F: BDOS_2FH_ReadLogicalSector()
0x30: BDOS_30H_WriteLogicalSector()
```

See BDOS documentation for details about the input/output registers.
These commands are no longer supported in Firmware version 4.0, but support is planned to be re-added.

DISKROM Commands

A list of commands supported by the host is available below, **fixed values are printed in bold**, *fieldnames are in italic*, finally registers (AF, BC, DE, HL) and flags CF, ZF, PF are written in capital letters. I.e. **AF** refers to a literal hexadecimal value of 175 (decimal) and AF is the combined accumulator + flag register.

Command name:	DSKIO
ID byte:	0x80
Description:	Service for diskrom DSKIO (@ $4010) requests, can read and write sectors directly from and to disk images on the host.

A =	Drive number relative to the diskrom, currently only 0 and 1 are supported, 0 = first nowind disk image, 1 second nowind disk image.
CF =	the carry flag is set to write sectors, otherwise read sectors
B =	Number of sectors
C =	0x80-0xFF: values in this range are media descriptors, the value of C determines the type of physical media to read, for nowind we always read from disk-images, so it is ignored. 0x00-0x8F: this range was never used for floppydisk media, it is used for FAT16 support on MSX harddisks. The lower 7 bits of C are used as the highest 7 bits of the start sector. (but only if C is in this range)

DE = start sector (lower 16 bits)
HL = transfer address

The start sector is a 23 bit value formed by the C, D and E registers.

`C & `**`0x7F`**	`D`	`E`

startSector = DE;
if (C < 0x80) startSector += (C << 16);

Possible 'Host -> MSX' Replies to the DSKIO(0x80) command

We should redesign this interface to make it simpler and re-use the generic 'transfer-data-to-memory' routines we have yet to implement!

~~**1)** Host to MSX transfer~~

~~More data available response (**AF 05 00**), the first 3 bytes are fixed values, followed by the transferaddress and the amount of data.~~

~~Byte~~	~~0~~	~~1~~	~~2~~	~~3~~	~~4~~	~~5~~	~~6~~
~~Value~~	~~**AF**~~	~~**05**~~	~~**00**~~	~~*ta_low*~~	~~*ta_high*~~	~~*amount_low*~~	~~*amount_high*~~

~~Followed by the amount of bytes specified with amount_high*256+amount_low and terminated by the **AF 07** trailer. These last two trailer bytes should be send back by the MSX and are used to verify the transfer was successful. See *Hardware Design Decisions* for details.~~

~~Byte~~	~~0~~	~~1~~	~~2~~	~~3~~	~~…~~	~~amount-1~~	~~amount~~	~~amount+1~~
~~Value~~	~~..~~	~~..~~	~~..~~	~~..~~	~~..~~	~~..~~	~~**AF**~~	~~**07**~~

~~**2)** Transfer backwards (faster, but only possible at even addresses)~~

~~Byte~~	~~0~~	~~1~~	~~2~~	~~3~~	~~4~~	~~5~~
~~Value~~	~~**AF**~~	~~**05**~~	~~**02**~~	~~*ta_low*~~	~~*ta_high*~~	~~*amount*~~

~~Here the transferaddress (ta) should be one byte past the last address written and amount is specified in blocks of 64 bytes. For example a transfer of 0x1000 (64x64) bytes to 0x7000 would be:~~

~~Byte~~	~~0~~	~~1~~	~~2~~	~~3~~	~~4~~	~~5~~
~~Value~~	~~**AF**~~	~~**05**~~	~~**02**~~	~~**00**~~	~~**80**~~	~~**40**~~

~~**3)** End of receive-loop (**01**) and no more data (**00**).~~

~~**AF**~~	~~**05**~~	~~**01**~~	~~**00**~~

Hardware Design Decisions

The FT245 USB IC that is used in the Nowind design can provide reliable transport of data using USB bulk transfers. These transfers are guaranteed by the USB protocol and any failed transfers will be retried automatically without the user knowing. **BUT** a hardware design decision we made to keep the design of the Nowind Interface simple re-introduces the need to verify the result of the transfer.

The Nowind Interface uses an FTDI FT245R, see for complete datasheet www.ftdichip.com.

Relevant specifications from the FTDI datasheet:

FIFO RX Buffer (128 Bytes).
Data sent from the USB host controller to the FIFO via the USB data OUT endpoint is stored in the FIFO RX (receive) buffer and is removed from the buffer by reading the contents of the FIFO using the RD# pin. (Host to MSX transfers).

FIFO TX Buffer (256 bytes).
Data written into the FIFO using the WR pin is stored in the FIFO TX (transmit) Buffer. The USB host controller removes data from the FIFO TX Buffer by sending a USB request for data from the device data IN endpoint. (MSX to Host transfers)

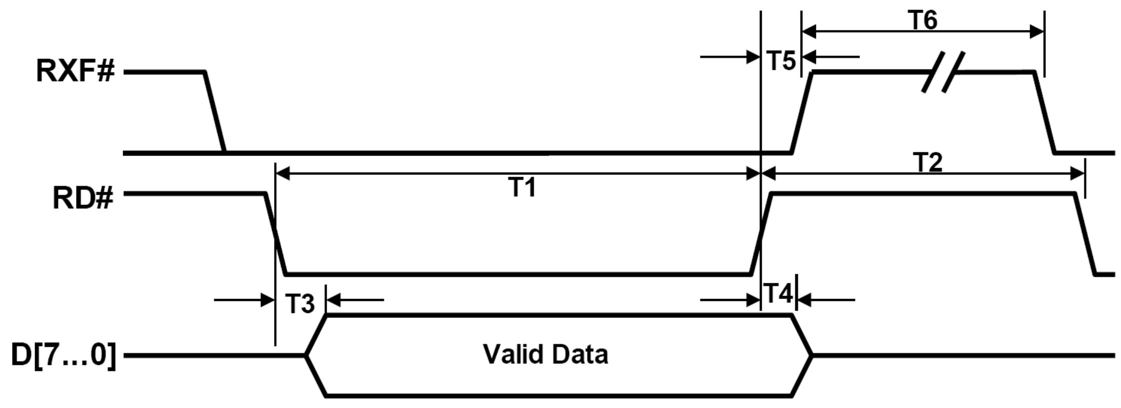
Reading data (transfer data from host -> msx)

The FT245 is connected in such a way that it is possible to 'miss' data when reading and to 'loose' fifo synchronization inside the FT245R when writing. *This sounds bad but is actually the reason why the design is simple and affordable* ***and*** *the transfer speed is maximized!* Ofcourse we do not want an unreliable Nowind Interface ;) so we do some extra checking in MSX <-> HOST protocol to verify the transfer results.

When the MSX reads data from the Nowind Interface and no data is available in the FT245's FIFO, the read command should never actually reach the FT245, if it does, the internal FIFO read pointer will be corrupted in such a way that it does no longer points to the first data received from the host. We want to able to poll for incoming data, but when we do, we risk corrupting the internal FIFO read pointer; we prevent this by using 'data available' signal (RFX#) to cancel out the RD# signal.

This way, the internal FIFO read pointer is never out of sync with the actual incoming data from the host. However writing is a different matter.

FT245R FIFO READ Timing Diagrams



Notice that RXF# and D[7..0] are output lines here and RD# is an input.

Time	Description	Minimum	Maximum	Unit
T1	RD# Active Pulse Width	50	-	ns
T2	RD# to RD# Pre-Charge Time	50 + T6	-	ns
T3	RD# Active to Valid Data*	20	50	ns
T4	Valid Data Hold Time from RD# Inactive*	0	-	ns
T5	RD# Inactive to RXF#	0	25	ns
T6	RXF# Inactive After RD Cycle	80	-	ns

If you look carefully at the data above, you will notice, that T1 and T2 are actually requirements of the FT245R we must meet and T3, T4, T5 and T6 and guarantees we get.

The signal _RD from the MSX we use to drive RD# has a low time will refer to as R1 and a high time R2. At first glance R1 needs to be at least 50ns (T1) and R2 50+80=130ns (T2). Because we have used RXF# to cancel out RD#, there are a few extra things to keep in mind.

T2 will always be high while RXF# is high!, so because T5 can be 0 and T6 is at least 80ns, an R2 of 130ns will cause a T2 of at least 130+80 = 210 ns! This also means an R2 of 50ns would cause a T2 of at least 50+80 = 130ns which already meets the requirements for T2.

If we would use R1 = 50ns and R2 = 50ns, T1 would be shorter then required, because RXF# (that stays high for 80ns) would completely mask the time of R1. This means, when using a R2 of 50ns a minimum R1 of 80+50ns is required. To summarize; read requirements:

R1 (_RD low time) > 130ns
R2 (_RD high time) > 50ns

Valid data can be read 50ns (T3) after _RD goes active (low) until _RD is released.

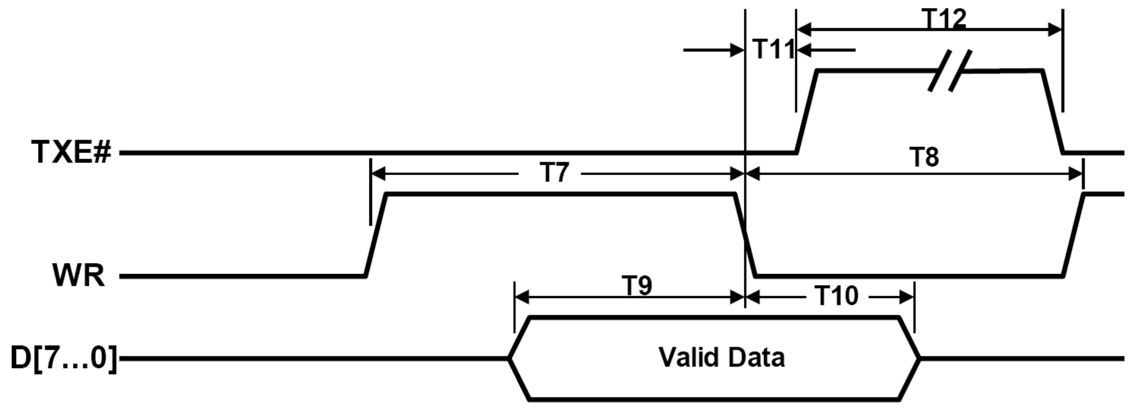
In the case a MSX' Z80 connected to this design, a Z80 @ 7.14Mhz has a T-cycle of 140ns.
A memory-read cycle takes 3 T-cycles, during which RD is low for 2 T-cycles (280ns) and high for 1 T-cycle (140ns). Also the data is sampled 2 T-cycles after RD goes low. This means the read timing requirements can never be broken by the Z80.

Writing data (transfer data from msx -> host)

The same synchronization problems (corruption of the FT245R's internal write pointer) can occur when writing data to the FT245 when the write-FIFO is full! Even if writing at the wrong time (when the buffer is full) would not corrupt the FIFO-write pointer, the data would still be lost, so connecting the TXE# to prevent WR signals **would not** solve this problem!

In short: both reading and writing isn't guaranteed to succeed, so we need the to verify the data transfers in both directions in the protocol-layer.

FT245R FIFO Write Timing Diagrams



Time	Description	Minimum	Maximum	Unit
T7	WR Active Pulse Width	50	-	ns
T8	WR to WR Pre-Charge Time	50	-	ns
T9	Valid data setup to WR falling edge*	20	-	ns
T10	Valid Data Hold Time from WR Inactive*	0	-	ns
T11	WR Inactive to TXE#	5	25	ns
T12	TXE# Inactive After WR Cycle	80	-	ns

Valid data must be present 30ns after the rising edge of WR (or earlier) and until at least the falling egde of WR.

Because _WR is (except for inversion) directly connected to WR, T7 and T8 must be > 50ns as expected.

In the case a MSX' Z80 connected to this design, a Z80 @ 7.14Mhz has a T-cycle of 140ns.
A memory-write cycle takes 3 T-cycles, during which WR is low for 1 T-cycle (140ns) and high for 2 T-cycles (280ns). Also the data is valid from before WR goes active until after WR goes inactive.
This means the write timing requirements can never be broken by the Z80.

Error occurrence use cases

1) If no data is available (the receive buffer is empty) the MSX will read 0xFF.
2) If data arrives at exactly the moment the MSX reads from the FT245, the first byte of data will be corrupted or lost. What happens here is that the MSX sends the read signal, samples the data bus and in the time between the sampling of the data bus and the rising of the read line, data arrives on the FT245. This means, the transition of RFX# will trigger a late RD# pulse. The FT245 responds by shifting the first byte of data out of the receive fifo, but since the MSX has already sampled the data bus, it is lost.
3) If data is written to the from the MSX to the FT245 and the send-buffer is full the data is not accepted by the FT245 (is lost) and even worse: the internal write-pointer of the FT245's FIFO is corrupted, this means any data written after this will be unreliable until the FT245 is reset. (or buffers are flushed?)
4) If data is read or written too fast (consecutive writes) for the FT245 to handle, that means, if the timing requirements are broken, data will be lost.

The first effect (1) or (3) occurs

- when the host is not sending/receiving data fast enough from/to the FT245 (ie. when the host CPU application is too busy to handle requests / 100% CPU usage?)
- when the host is not sending enough data (ie. protocol bug)
- if the MSX reads data faster then 12Mbit USB bus can provide (unlikely / not possible?)

Practically, the MSX cannot read or write that fast, even at 7Mhz, so this only happens when the host CPU is too busy to handle USB requests in time. (assuming we have no protocol bugs ☺)

The second effect (2) occurs:

- each time the FT245 receive buffer state changes from 'empty' to 'not empty', which is, every time the MSX waits for a response from the host and receives it.

The last effect (4) never occurs, the MSX read and write operations just are not fast enough even at 7,14 Mhz.

- ~~when the MSX executes read or write operations too close together, this can happen due to a 16bit data bus operations at 7Mhz. The 16bit operation consists of 2 very fast consecutive 8bit operations that are to fast for the FT245 to handle~~.

Notice that **only effect 2** is introduced by our decision to combine RFX# and RD#, which is technically speaking not correct usage of the FT245 because a read should only be initiated using RD# **after** RFX# indicates that data is available.

Protocol Solutions

Effect 1

The client/server protocol compensates for the aforementioned issues in several ways. Effect 1 cannot be prevented, but its occurrence can be detected. Effect 1 for data send from the host to the MSX is detected by sending two trailing bytes **AF 07** after the actual data block, these two extra bytes are send back to the host. The host checks whether is actually receives these bytes. When the effect 1 should occur the data stream will lose synchronization and one or more 0xFF bytes (buffer empty) will be read by the MSX. As a result the **AF 07** bytes will not be received by the host, the error is detected and the transfer retried.

Effect 2

For read operations (data from the host to the msx) Effect 2 is prevented by prefixing any data send from the host to the MSX with three special header bytes **FF AF 05**, where the first FF byte might be lost due to effect 2 and the **AF 05** bytes are used to re-synchronize the data stream.

This is not an issue for write operations, because the _WR signal of the Z80 is connected directly (except for inversion) to WR of the FT245. Write operations have other issues though, namely effect 3.

Effect 3

This is worked around by the used protocol, we transfer data in blocks that are smaller then the FIFO size of the FT245 and the hosts acknowledges each received block. This ensures the FIFO is never full.

~~**Effect 4**~~

~~Effects 3 must be prevented by not using 16bit read/write operations (at least at any clockspeed > 3,57 Mhz).~~

Todo: provide timing details from the FT245 datasheet to back this up.