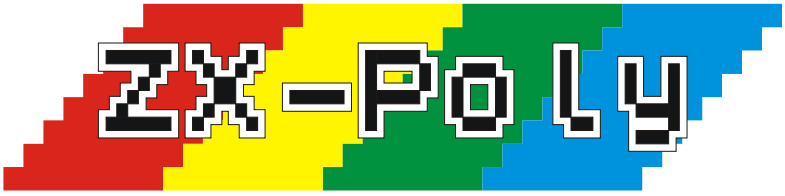
The multiprocessor system ZX-Poly™



Version: 2.10
Author: Igor Maznitsa (igor.maznitsa@igormaznitsa.com)
Date: 28 may 2011

Contents

The license agreement

All Copyrights by Igor Maznitsa (igor.maznitsa@igormaznitsa.com), All Rights reserved. The platform ZX-Poly is an intellectual property of its owner (Igor A. Maznitsa) and no one can implement, use or revise the platform for any commercial purpose without special signed permission of the owner. If someone wants to develop underlying ideas or make make any non-commercial implementation of the platform, the new development must contain information about the underlying platform (ZX-Poly) and the platform owner (Igor Maznitsa) must be notified.

The Foreword

A long time ago, in a galaxy far far away... the ZX-Spectrum platform was the most common home computer platform both in the World and in Russia (but in Russia we had got the platform actually lately for our technological lagging and the iron curtain). Unfortunately the platform had not been being developed quickly to follow increasing user requirements and it was lost in fighting with other platforms (the PC and Mac) pushed ZX from the market and user's homes (but not from hearts). In Russia we did have really other problems. Whole the USSR worked only for army (like the North Korea) and we didn't have any access to the modern world's developments (of course we saw them as a color pictures in our technical magazines but we didn't have any hope to get them in own use).

Fortunately our very advanced engineers were able to make the Russian clone of the ZX-Spectrum platform (at their free time of course, when they were not very busy by nuclear rocket development) on the element base which was accessible by citizens of the USSR through buying on non-legal electronic markets (in Russian it sounds like 'toulkuchka'). The Russian ZX-Spectrum platform variant was very fast and stable (and even didn't have some errors in its circuit which there were in the original platform) and looked very prettily on the background of other computer platforms being in usage on the former USSR territory, because all used computer platforms had very high price and low features (for instance, the soviet Apple II clone, called AGAT, had the same price as a very good car (or a good flat) in the end of 80th).

A lot of small home based companies (very often they were non-legal ones but the companies had very bombastic titles end main with “corporation” of course) began to develop the platform and software projects for it. Our enthusiasts had developed a lot of extra hardware and solutions for the platform - hard disks, sound cards, memory extenders (there was even a Spectrum clone with 1 megabyte RAM). But anyway the most serious problem of the platform was not solved – very restricted graphic possibilities chasing us all that time and any attempt to solve the problem made us loosing any back compatibility, that was unacceptable.

Well, I begin my own story. In 1989 I had met my first personal computer called BK-0010-01 and saw ZX-Spectrum computers only in amusement arcades (its games just amazed me, I was sure that it is just a super computer). BK-0010-01 had only 32 kilobyte memory and was able show only 4 colors for pixel (but for every pixel!) but ZX-Spectrum seemed really better in that case and ZX had 64 Kbt !!!! So that I had assembled my own first ZX-Spectrum in 1991. From 1990 till 1995 we had flourishing of the ZX platform, there were even companies (producing ZX-Spectrum clones) make sponsorship for well-known Russian TV shows. It was really the golden age of ZX-Spectrum in Russia..

In 1994 I was called up for military service but didn't forget the platform and in the middle of 1994 my mind was visited by the idea – how to extend the graphics possibilities of the platform and save the back compatibility. It was indeed very easy, genially and possible to be implemented in hardware! I began connect with well-known Russian ZX-Spectrum manufacturers since 1996 but unfortunately the “ZX-Spectrum golden age” had been passed away and I didn't get any support and understanding from their side and I was made to shelve the idea.

In 1995, I had bought my first PC computer (PC AT 486) and got access into the FidoNet (the russian part) as the 2:5030/153.31 point. At 1997 I had left the army and began work in commercial software development, I didn't forget my idea and still tried to bring it in minds, thus I wrote an article for the Russian ZX-Spectrum news group and title it “A color space extender for the ZX-Spectrum platform” (09 june 1999) but didn't get any support from the Russian community because the community had very orthodox point of view on the platform and didn't want any changes at all. Because I already was a skilled PC programmer, to check my idea I wrote a platform emulator and saw that the idea really works!!! I was interviewed by a Portuguese spectrum fan about my platform (which had the work name ZM-Polyhedron at that time), I didn't know English in that time at all and mainly used computer translators (it didn't work well too) so my interview looked a bit crazy :) (I am sure that Tarzan would speak in such manner too) but I made attempt to bring my idea in the big World. Lately I renamed the platform to ZX-Poly to make it shorten and good for ears.

Well, what is my idea shortly? It is very very very easy – if we have a few devices work simultaneously, executing the same program, based on the same oscillator and the reset signal source, we will get the same work result (it is very often is being used in military products to increase robustness of military systems), but it is not all, we can save the same command chain but change the data for every executing part and we will have the SIMD structure (single instruction many data). But how we can use it for ZX-Spectrum? The answer is we just can connect four ZX-Spectrum computers together through a special dispatcher and each will be processing its own color part (R,G,B,Y).

As the conclusion for the document part – the platform shows that we actually could have very strong and no so expensive home platform (not x86 based) in the end of 80th, which would have a lot of software and games and could be home assembled in third world countries, its possibilities would be not lower than IBM XT 286. And may be the idea could save the Sinclair Research Ltd in the home computer platform business until today.

The Common description

The ZX-Poly™ system is a multiprocessor platform which is developed in the symmetric scheme, it means that all its processors are identical each other in their data processing possibilities and machine command set (may be it would be good to use all CPU parts of the same factory, it may decreases possible problems). There are four CPU elements in the system. Z80 or any its clone is being used as a CPU element of the system. The system has the common memory field (it has RAM 512 kilobyte and the ROM size as the standard ROM size of the ZX-Spectrum 128 i.e. 32 kilobyte). As the OS I offer to use the standard ZX-Spectrum 128 OS and as disk OS I use TR-DOS.

The Block diagram

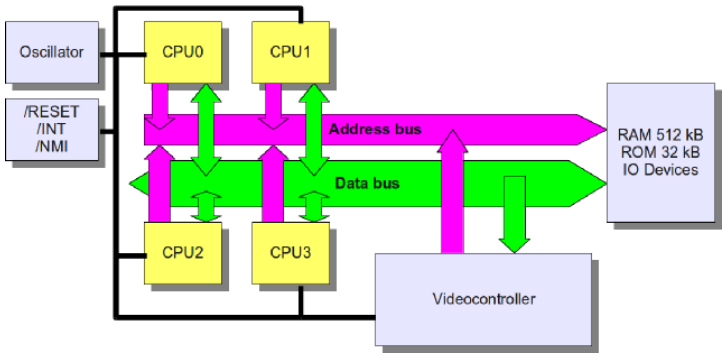


Image 1: The block diagram

You can see on the diagram all main parts of the platform. The base of the platform – the CPU module group contains four CPU modules (CPU0-CPU3) which fulfill all main computing and managing functions in the system. On the right you see the common RAM field and the ROM field, they are common for all those processors (!). The video controller forms the video signal (to be viewed on TV) from the data found in the common memory field.

The Common technical description

On the architecture layer, the platform just adds a few new IO ports into the standard ZX-128 port list (as all other platforms based on the ZX-Spectrum architecture). There are the main managing #3D00 port and a few ports in the #<00-FF>FF interval. All extensions which are being brought by the platform can be blocked by an application to increase back compatibility in any time but the blocking will be removed only by a full system reset. A CPU system part (module) contains the Z80 compatible CPU, the ZX-128 managing register and a few registers for the ZX-Poly™ mode. The CPU marked as the CPU0 has more rights and has a higher priority against other CPU modules because the CPU module is managing the system and can configure work processes of other CPU modules (CPU1, CPU2, CPU3).

The priorities in the system

The platform implements the primitive “stair” priority system. The CPU0 is the main module for the system, thus it has the highest priority and it only can write into the system configuration port #3D00.

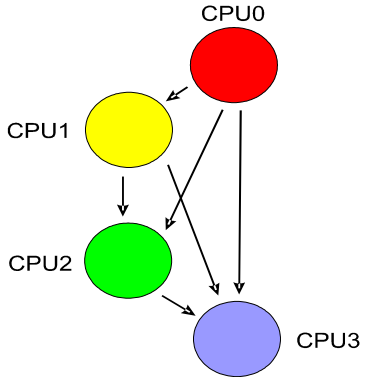


Image 2: The priority diagram

Mainly the priorities are needed to protect inside registers of a CPU module against any changes by other CPU module, i.e. the CPU module registers can be changed only by either the CPU installed on the module or a CPU-module with higher priority. I made the thing to give a possibility to organize ring in OS executing. The CPU0 has the highest priority and the CPU3 has the lowest priority. But you should keep in your mind that the priority system doesn't restrict reading operations but only writing operations (!), it means that CPU3 can get the information placed in configuration ports of CPU1 but can't make any changes.

The configuration port ZX-128 (#7FFD)

The port has been descended from the standard ZX-Spectrum 128 and it has a few added functions in the ZX-Poly™ platform. The port is presented on each CPU module and can be changed only by the module installed CPU unit, any other CPU module (and CPU0) can't change the local #7FFD port, it is a private module port!

INTCPU0	ROMRAM	Lock	ROM	Screen	RAM2	RAM1	RAM0
7	6	5	4	3	2	1	0

Destination of the port's bits:

- Bits 0,1,2 – they contain the index of a RAM page contained in the CPU page 3 memory area (from 0 to 7).
- Bit 3 – it contains the video RAM page number that contains the data for the video output. If the bit is reset, the RAM5 will be used as the video data source, else the RAM7 will be used.
- Bit 4 – it contains the ROM page number mapped on the #0-#3FFF memory addresses. If the bit is reset, the ROM0 will be switched on (the BASIC-128 extension), if the bit is set, the ROM1 will be presented in the memory area (the BASIC-48).
- Bit 5 – if the bit is set, all data manipulation with the port will be blocked on the hardware level and the bit can be reset only through full-system RESET.
- **Bit 6** – it is actually new one and used by the ZX-Poly™ platform. If the bit is reset, it will map the selected ROM page into the #0-#3FFF memory area. If it is set, the RAM0 page will be used instead of any ROM page in the memory area. All bit manipulations over the bit will be blocked if either the #3D00 port or the #7FFD port is blocked (the 5th bit is set), the new bit value will be just ignored. Also you should keep in mind that since the #3D00 port is blocked, any attempt to write data into the memory area #0000-#3FFF will be ignored even if

the RAM0 is presented there instead of a ROM page. It allows to simulate own "ROM" images.

- **Bit 7** – it is new ZX-Poly bit too and **used only by the CPU0 module** (!). If it is set, the system INT signal will be blocked and the CPU0 will not get it until the bit reset. When the port #7FFD is blocked (the 5th bit is set), the state of the bit will be ignored and the INT signal will not be blocked but will be working as on the standard ZX-Spectrum. The bit is used only by the CPU0 module and can be changed by the CPU0 only (and the effect will be only for the CPU0!).

As you see the port has own lock bit so that the port can be written even when a module has blocked writing operations into out ports, in other words, if you disable writing into output ports (take a look at 4th bit of ZX-Poly R0) the #7FFD port will be accessible anyway. It allows to increase compatibility and make old software adaptation much easier.

The ZX-Poly™ main configuration port (#3D00)

The port is the "heart of the system" and the main configuration port for the ZX-Poly™ system. All CPU modules management and video mode choosing pass through the port. I have chosen the port number because:

- The port address is not used by the standard ZX-Spectrum platform.
- The port address 15616 is similar to the TR-DOS entry point and it allows to use the same address decoder.
- The port being mapped on memory is placed in the ROM address area, it allows the CPU0 module to manipulate the memory area of an other processor module (which is being mapped as an IO module) without any threat to inter-cross with the port address. This port is the only port allowed to be read and write by the CPU0 when the "other CPU module memory as the IO space" mode is active (you will find more information about the mode below).

You must keep in mind that if you lock all port output operations for the CPU0 module, this port will be blocked too and it wont be possible to change its state until total system reset.

For writing

As an output port the #3D00 port allows to manage both the current system state and the current video mode. All bits of the port in use.

BLOCK	CPU IO1	CPU IO0	Video2	Video1	Video0	RESET	/WAIT
7	6	5	4	3	2	1	0

The destination of the port bits:

- Bit 0 – if the bit is reset, the WAIT signal of CPU1-CPU3 modules will be active and all that modules will be in the sleep mode. Default value of the bit after a system reset is 0 so that all CPU modules (exclude CPU0) is sleeping (!).
- Bit 1 – if the bit is set, all (!) CPU modules (CPU0-CPU3) will get one RESET signal on their pins, the signal will be sent only on the CPU chips (!), other system devices will not get the reset signal (!). After the signal, each CPU in the system will read the first command not from memory but from inside ZX-Poly registers of its module. If you set this bit, the 0th bit of the register will be set automatically (all CPU modules will be started simultaneously!).
- Bits 2,3,4 - These bits set the current video mode for the system, it will be described further in the document.
- Bits 5,6 – These bits contain the index of the CPU module which memory will be mapped to the IO space of the CPU0 module and each IO out/in operation of the CPU0 will be making writing/reading in/from the memory cell that has the same address as the port address in the operation. These bits are used only by the CPU0 module! After an out operation, a /NMI signal will be sent to the CPU chip of the IO mapped module and after a reading operation,

an /INT signal will be sent to the CPU. If all the port bits are reset then the mechanism will be deactivated (as you see the situation also shows that CPU0 can not make itself as an IO mapped module). If any bit from these is set, the selected CPU module will be selected by the platform as the IO module for the CPU0 module. But pay your attention that the #7FFD and #3D00 ports are active and the CPU0 can't write into the RAM #7FFD or #3D00 addressed cells of a mapped CPU module!

- Bit 7 – the bit locks the port. If you set it, all (!) ports of ZX-Poly™ (I mean additional ports) will be locked and the state wont be changed until a full-system RESET signal and the platform will be working in the most compatible mode with the standard ZX-Spectrum platform. Since the port is locked you will not be able to block system INT and NMI signals for all CPU modules (exclude the CPU0 module), writing operation into inside module registers will be locked (!). Also the interruption signal generation when the HALT command is found, will be locked too.

After a full-system RESET signal, each bit of the port has 0.

For reading

The #3D00 port not only can be written but contains very important information for read. The port is non a symmetrical one for read-write operations and you can not read the written data via a read operation. Each processor module reads its own (!) local value of the #3D00 port and the value depends on both the module configuration and the module index.

RAMoff2	RAMoff1	RAMoff0	IOforCPU0	MEMdisabled	IOdisabled	CPUindx1	CPUindx0
7	6	5	4	3	2	1	0

The destination of the port bits:

- Bits 0,1 – the bit pair shows the index for the processor module, in other words, the 0 value shows that the module is the CPU0 module and the 3 value shows that the module is the CPU3 one.
- Bit 2 – The bit shows that IO port writing operations is disabled for the module by the main CPU module, in other words, any OUT operation will be executed but its written data will be ignored (like NULL device).
- Bit 3 – The bit shows that memory writing operations is disabled for the module by the main CPU module, in other words any memory writing operation will be executed but the memory cell is being used in the operation will not be changed and written data will be just ignored.
- Bit 4 – The bit shows that the CPU module memory is being mapped as the IO port map for the main CPU module (CPU0) and all port operations of the main module are changing or reading the memory data of the IO module memory.
- Bits 5,6,7 — The bits show the current memory offset (in memory pages not as cells) where the module memory area is being started (a memory page has 64 KByte) in the common memory area.

The available video-mode table

№	Bit 4	Bit 3	Bit 2	Description
1	0	0	0	The standard ZX-Spectrum 128 video mode, the CPU0 memory space is the video data source.
2	0	0	1	The standard ZX-Spectrum 128 video mode, the CPU1 memory space is the video data source.
3	0	1	0	The standard ZX-Spectrum 128 video mode, the CPU2 memory space is the video data source.
4	0	1	1	The standard ZX-Spectrum 128 video mode, the CPU3

				memory space is the video data source.
5	**1**	**0**	**0**	**The ZX-Poly low-resolution video mode. The video mode has got the 256x192 pixel resolution. All CPU memory spaces are being used as the video data source for the video mode. The CPU0 video memory provides the R (red) component for a pixel, the CPU1 video memory provides the G (green) component for the pixel, the CPU2 video memory provides the B (blue) component for the pixel and the CPU3 video memory provides the BRIGHT component for the pixel. The video mode allows to preserve the most possible compatibility with the ancestor-platform and show old software in the rich color video mode when each pixel can get one from 16 colors (not a 8x8 block!). The video mode doesn't use any video attribute memory data and their values are just ignored. I had ideas to use the attribute values to increase graphical possibilities (as an example they could be used as a palette) but I gave up those ideas because it would increase complexity of implementation too much.**
6	**1**	**0**	**1**	**The ZX-Poly high-resolution video mode. In the video mode you can see on the screen the 512x384 pixel field. All memory spaces are being used in the video mode as video data providers. The video data from all cpu modules is combined in the chess-order and outed on the monitor as 2x2 pixel blocks. The video mode us using the color attribute information in the same way as the standard ZX-Spectrum. A four pixel block has the following order: the top-left pixel – provided by the CPU0 module memory space, the top-right pixel – provided by the CPU1 module memory space, the bottom-left pixel – provided by the CPU2 module memory space, the bottom-right pixel - provided by the CPU3 module memory space.**
7	1	1	0	The combination is reserved and behavior in the state is not defined.
8	1	1	1	The combination is reserved and behavior in the state is not defined.

How it works...

A Full-system RESET signal

A full-system RESET signal is being sent to the system after a user pressing the RESET button. During the signal processing all ZX-Poly™ ports will be written by 0, and the ancestor-platform ports (ZX-128) will be written by values according the ZX-Spectrum 128 standard. You see that CPU1-CPU3 will be slept (because the WAIT signal will be active) after the full-system reset. The video controller will be configured to work in the standard ZX-Spectrum 128 video mode and the CPU0 memory data will be being used as the video data source.

A Processor module work process

Each processor module has several own local registers those can be reached by the module CPU via its IO operations. Also each processor module has own local #7FFD port used by the ZX-128 architecture. To manage work in the ZX-Poly™ mode, there are four eight bits registers placed

on each module. The registers can be locked in their current state, if the #3D00 port is locked through 7th bit set. The locking state can be reset only with a full-system RESET signal (by the RESET button). In the normal state (if the #3D00 port is not being locked) system standard interruption signals (NMI and INT) are being sent only to the CPU0 module but when the 7th bit is set, those system signals will be being sent to all (!) CPU modules. Data saved in the inside registers can be changed by either the CPU unit of the module or by the CPU module that has a higher priority (as you remember the priority of the CPU0 is more than the priority of the CPU1, CPU0>CPU1>CPU2>CPU3). The ports is accessible through IO operations and their addresses for each module can be calculated by the formula below:

([module_index] << 12) | ([register_index] << 8) | #FF

The ZXPoly R0 register

The register is only local register allows both read and write operations. Each processor module has access to read the port but only either the own module or a higher priority module can write data in the register. The port is non-symmetric one and you can't read the same data which you have written in.

For writing

The register manages the current processor module work.

INT	NMI	RESET	OutDis	MemWrDis	PageOff2	PageOff1	PageOff0
7	6	5	4	3	2	1	0

The destination of the register bits:

- Bits 0,1,2 - these bits contain the RAM page number which is indicated by the platform as the first memory page for the module in the common memory space. Every page has the 64 KBt size. Be careful! The mechanism allows to intersect memory areas (fully or partially) of CPU modules and it is possible to make all modules work in the same memory space! All addresses are being used by memory operations of the module will be remapped with the information of the bits. Since a full-system reset the bits contain state that depends on the CPU module index and the state is calculated by the formula: **[module_index] << 1**. As a few processor modules are trying to change the same memory cell in the same time, the operation result (and the memory cell data) is not defined because all CPU modules are working simultaneously (the race problem will be detected).
- Bit 3 – The bit disables memory writing operations for the module, in other words all data changes will be ignored and it looks like to write data on NULL device.
- Bit 4 – The bit disables IO port writing operations. The bit works in the same way as the 3th bit but has effect for IO writing operations instead memory.
- Bit 5 – The bit allows to send local RESET signal on the module CPU unit. After the reset signal the next three bytes will be read not from memory (!) but from inside module registers! The mechanism allows you to make jump to any address.
- Bit 6 – The bit sends a local NMI signal to the module CPU unit.
- Bit 7 – The bit sends a local INT signal to the module CPU unit.

For reading

The module register can be read by and CPU module and the register allows to get current module work state.

ADDR15	ADDR14	ADDR12	ADDR8	ADDR2	ADDR1	WAITMode	HALTMode
7	6	5	4	3	2	1	0

- Bit 0 – The bit shows that the CPU has met a HALT command in the command flow and is being stopped until an event.

- Bit 1 – The bit shows that the module CPU is being in the WAIT mode. It can be for either the stop-address mechanism or for the common wait CPU1-CPU3 mode just after common system reset.
- Bits 2-7 – The bits show the state of 1,2,8,12,14,15 bits of the last command address that was placed on the address bus by the CPU unit during the first command byte read (M1 was active). Pay your attention that the address relates to the first byte of a command, if the command is a multibyte one it shows the first command byte address bits. Mainly the bits allow you to understand the PC register value (not very precisely but anyway) of the module CPU unit.

The ZXPoly R1 register

In opposite to the ZXPoly R0 register, the R1 register (and others) can't be read through IO operations and is accessible only for writing. The register has two destination modes:

- to manage the module
- to contain the first read command byte after a local module RESET signal (given through 5th bit of the module R0).
- If the module cpu in halt mode and you send a NMI signal (through 6th bit of the R0 register) the 4 lower register bits will be placed on the cpu data bus during NMI signal to be used by the cpu.

For writing

HaltNMI	HaltINT	Dis7FFDio	DisNMI	CPU3	CPU2	CPU1	CPU0
7	6	5	4	3	2	1	0

The destination of the register bits:

- Bit 0 – The bit shows that the CPU0 must get notifications about met HALT command.
- Bit 1 – The bit shows that the CPU1 must get notifications about met HALT command.
- Bit 2 – The bit shows that the CPU2 must get notifications about met HALT command.
- Bit 3 – The bit shows that the CPU3 must get notifications about met HALT command.
- Bit 4 – The bit is block any NMI signal for the module CPU unit. It just switch off the cpu pin from the signal
- Bit 5 - The bit is being used only by the CPU0 module in an IO mapping mode (when other module is being mapped for IO operations of the CPU0 and instead ports CPU0 writes into the module memory and read too). If the bit is set, the CPU0 is not be able to change own #7FFD port and the data for the port will be written in the memory cell #7FFD of the mapped IO module.
- Bit 6 – The bit sends an INT signal on the CPU signed by 0,1,2,3 bits if the HALT command is met. **Be careful and keep in your mind that CPU module can notify itself!**
- Bit 7 — The bit sends NMI signal on the CPU signed by 0,1,2,3 bits if the HALT command is met. **Be careful and keep in your mind that CPU module can notify itself!**

As the #3D00 port is locked, the system will not be supporting the notification mechanism to increase its compatibility with the ancestor-platform (ZX-128). The 4th bit disabling NMI signals is very important because it allows to switch off the NMI signals generated during the CPU0 works with its address space through IO operations (it is being used to simulate an IO devices by CPU1-CPU3 modules), also NMI signals can be switched off by the DI command of course.

The ZXPoly R2 register

As the R1, the R2 is accessible only for writing too. But it's behavior is much simpler because the register mainly is being used to save either the seconds command byte for local RESET signal processing or the stop address low byte. The stop address is very important point for the platform because it allows to synchronize executing flow among cpu modules. In a nutshell, the stop address is a just PC register value which is being compared with the first CPU command byte

address and if it equals then the CPU will get into WAIT mode (you can read about the subject below).

For writing

A7	A6	A5	A4	A3	A2	A1	A0
7	6	5	4	3	2	1	0

A7,A6,A5,A4,A3,A2,A1,A0 contain bits that correspond to the CPU address bus lines with the same names.
WARNING!!!! The stop-address mechanism works only if the #3D00 is not locked (to increase compatibility)!!!

The ZXPoly R3 register

It works like the R2 and just contains either the third command byte for local RESET mode or the stop-address higher byte.

For writing

A15	A14	A13	A12	A11	A10	A9	A8
7	6	5	4	3	2	1	0

A15,A14,A13,A12,A11,A10,A9,A8 contain bits that correspond to CPU address bus lines with the same names. About the stop-address mode, read further.

The System RESET

There is only way how it can be happened – the RESET button pressing by a user. The system RESET inits all system modules and IO devices. All bits of the #7FFD port are reset. The 3-7 bits of the ZXPoly R0 are reset to 0 but 0-2 bits are set in accordingly the expression:

[module_index] << 1

(I repeat, a memory page in the system has 64 Kbt size, it is not the same as a 16 Kbt memory page of the standard ZX-128). Pay your attention that after full-system reset, all inside command byte counters of processor modules contain 0 (so modules will not be loaded from inside command registers). Also, after a full-system reset, the stop-address mechanism is blocked for the first command to avoid any conflict situation because all inside registers contain zero and their value can be interpreted as a stop-address.

The Local RESET

The local RESET signal works locally for each module and doesn't have any effect to others, it's absolute local signal. The signal places 3 into the module inside command byte counter. The counter will place three command bytes from ZXPoly R1,R2, R3 sequentially and the CPU will be getting the bytes instead memory cell values. It allows the CPU to make jump on any memory address. Pay your attention that the registers content will be placed on the data bus even if there is an IO command in the registers, in other words the data bus will be exclusive kept until the counter is zero. After the counter is being reset in zero and the last command byte has been placed on the bus, all ZXPoly R1, R2, R3 registers will be reset in zero (!) to avoid any conflicts. If you have NOP, NOP, NOP in those registers, the CPU will start its work with the memory cell data since the #0003 memory address, 0,1,2 memory cells will be just ignored in the local reset case.

The Stop-address

The stop-address is a feature of the platform (may be the only) which allows to make a join for command threads executing by CPU modules. In a few words, you just define an address

which will move a CPU module in a wait mode if the CPU will try to execute any command on the address. The feature can be used only (!) if the #3D00 port is not being locked. Below you can see conditions:

- The CPU has activated its M1 signal.
- Inside local reset command counter is zero.
- #3D00 is not locked.
- The CPU has placed the same address on the address bus as the address has been saved in the ZXPoly R2 (A0-A7) and ZXPoly R3 (A8-A15)

If conditions above are satisfied, then WAIT signal will be set for the module CPU. The WAIT can be removed only if to change the stop-address (through ZXPoly R2,R3), or make either system reset or local reset for the module. But you should keep in mind that there is the 0^{th} bit if the #3D00 port and if the bit is reset, then CPU1-CPU3 are being in the wait mode.

How to organize work with IO ports

Each CPU module has its own IO ports listed below:

№	Port address	For readng	For writing	After a full-system RESET	Description
1	#7FFD	--	Y	0	It is the standard configure port of the ZX-128 platform, ZX-Poly™ uses unused bits of the port.
2	#3D00	Y	Y	module_index << 1	It is the main conduct port for ZX-Poly™, it can be written only by the CPU0 module but for reading it is accessible by each module (keep in mind that it has specific read data for each processor module).
3	#X0FF	Y	Y	0	It's a local module port ZXPoly R0. It is accessible to read by each processor module but can be written only by the same module or by the module with higher priority.
4	#X1FF	--	Y	0	It's a local module port ZXPoly R1. It can't be read but can be written only by the same module or by the module with higher priority.
5	#X2FF	--	Y	0	It's a local module port ZXPoly R2. It can't be read but can be written only by the same module or by the module with higher priority.
6	#X3FF	--	Y	0	It's a local module port ZXPoly R3. It can't be read but can be written only by the same module or by the module with higher priority.
7	Other	Y	Y	--	All other ports are accessible to be read/write without any systen restrictions but it depends on local module policy which is tuned through registers.

The memory heap usage

The memory map after the system RESET

The memory map

Since a system RESET signal (which as you may be remember is providing through the RESET button), there are four memory windows in the common memory space (heap), a window per a CPU module. The full memory heap size is 512 Kbt (128 * 4). Every module has its own 128 Kbt window in the memory heap and just after start the windows are not intersected. It is possible to move the memory window of a module through change local module ZXPoly R0 register. Inside a window, the memory is divided in 8x16 KBt pages like the standard memory configuration of the ZX-128. The top of the each memory window (the offset #0000 to #3FFF) has the ROM data, the ROM page for every module depends on the module #7FFD port state.

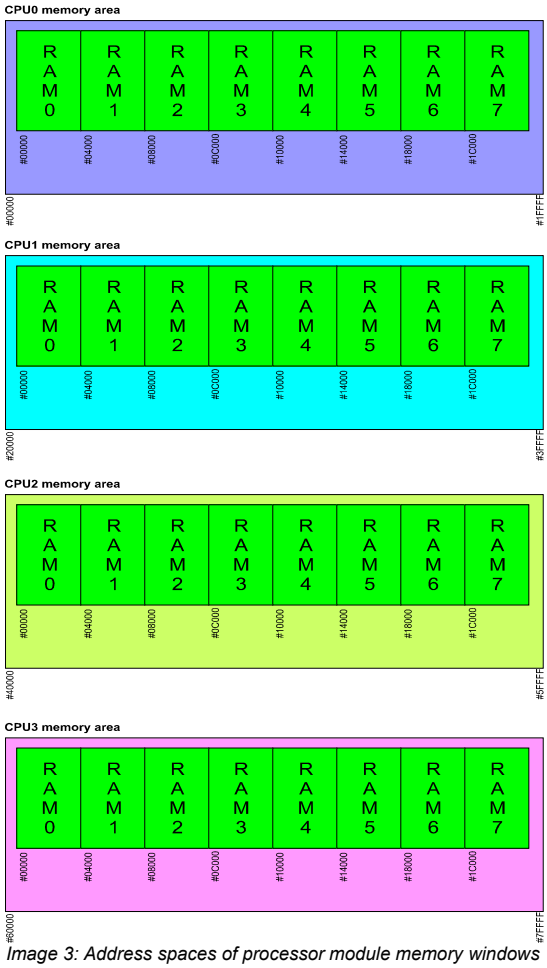


Image 3: Address spaces of processor module memory windows

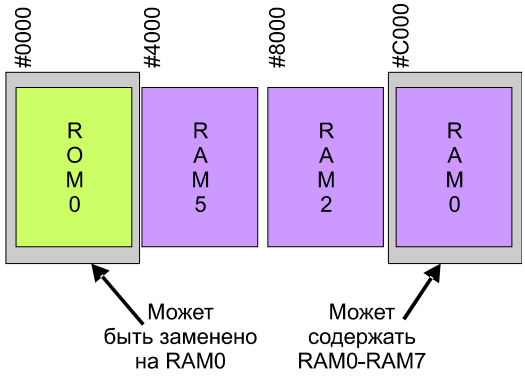


Image 4: The address space of a processor module.

Just after a system RESET, there will be the ROM0 page in the memory area offset #0000-#3FFF, but the ROM page can be changed to the RAM0 page (of the module) if to set the 6th bit of the module #7FFD port. When the #3D00 port is not locked and the RAM0 is mapped in the ROM position, you will be able to change memory data in the #0000-#3FFF area but if the #3D00 port is locked, any attempt to write into the #0000-#3FFF area will be rejected (the cell data will not be changed, the ROM behavior). If the #7FFD port is locked, the #0000-#3FFF will contain selected ROM page and the 6th bit state will be just ignored to keep full compatibility.

How to calculate the heap address from the logical address

As I write above, the common memory heap of the system consists of 512 Kb RAM solid block. Because a Z80 processor can address only 64kB, so we have to organize logical mapping of addresses. The mechanism extends the address placed by a module CPU on the address bus, the system adds special bit combination from the ZXPoly R0 module register and the address is being increased to 18 bits. The full 18 bit address is shown below:

A18	A17	A16	A15-A0
PageOff2 (ZXP R0.2)	PageOff1(ZXP R0.1)	PageOff0(ZXP R0.0)	Z80 A15-A0

As the first, the system calculates the physical memory address as written above but then it uses 0-2 bits of #7FFD those contain the offset of a memory page placed in the logical #C000-#FFFF memory offset to be compatible with the standard ZX-128 memory system.

The system ROM

As the system ROM we the standard ZX-Spectrum 128 SOS can be used. The size of the ROM supported by the system is 32 KB and it is divided to two 16 KB pages. The ROM pages accessible by a module can be switched through manage of the #7FFD local (!) module port, as you see, each module can work with own ROM configuration. In the standard ZX-Spectrum 128 the ROM 0 page (#0000-#3FFF) contains the SOS ZX-128 and the ROM 1 page (#4000-#7FFF) contains the Basic Sinclair 48. Just after system RESET, the ROM1 page will be switched on.

The video memory

It is the memory areas which provide color data to form video-signal. In opposite to the standard ZX-128, the ZX-Poly has six color video modes with different features.

Video-modes 0,1,2,3 (Standard ZX-128)

These video modes work like the standard ZX-Spectrum video mode, in other words they haven't any different exclude the possibility to change the source video memory areas. Each mode have the 256x192 pixel resolution and (like standard) two attribute colors for every pixel, also the mode uses the non-linear ZX-Spectrum memory addressing.

In the mode 0, the video controller uses the CPU0 video memory area, the mode 1 uses the CPU1 memory, the mode 2 uses the CPU2 memory and the mode 3 uses the CPU3 memory. The attribute format for the modes works like the standard ZX-128 attribute format:

- Bits 0-2 the ink color index (0-7).
- Bits 2-5 the paper color index (0-7).
- Bit 6 switches on the brightness mode (for both the ink color and the paper color)
- Bit 7 switches on the flush mode, color indexes of the ink color and the paper color will be replacing each other with the 2 Hz frequency.

When the system works in the modes, a programmer is given by very powerful possibility to organize a multicolor effect because instead two graphic pages on the ZX-128, in the case it is possible to operate with eight (!) video pages from different processor modules (potentially) as well as it is possible to switch the video modes and use the “hidden” video memory areas as an additional memory (may be 6kB is no such big size but..).

Image 5: The screen-shot shows screen in 0,1,2,3 videomodes

The video-memory for the video modes can be placed in either the RAM5 page or the RAM7 page, like ZX-128 it is being selected through the #7FFD port (if it is not locked of course).

The video-mode 4

Hurrah! It is just fantastic! The video-mode is the dream of a few spectrum user generations!

The video mode has both the same 256x192 pixel resolution and the same non-linear ZX-128 addressing but in opposite to the standard ZX-Spectrum video mode, it doesn't use attribute color data and doesn't have the 8x8x2 color restrictions. The mode uses video information from ALL CPU MODULES (CPU0-CPU3) simultaneously and the video controller combines the information to calculate pixel color. The CPU0 module provides information for the RED component (R), the CPU1 module provides the GREEN component (G), the CPU2 module provides the BLUE component (B). You may ask me - “and where is the CPU3?!” Don't worry, it is being used too, the CPU3 processor provides the Y (Bright) component in the result color. As you see the color combinations reflects the same colors as in the standard ZX-128 but FOR EVERY SCREEN PIXEL! The color RGB index is calculated through the formula (it's not e=mc^2 but near):

(Bit_G << 2) | (Bit_R << 1) | Bit_B

In other words, the video mode contains four layers (Red, Green, Blue and Bright) and each CPU module provides data for its own layer. As I write before, attribute data is not used in the mode at all. The mode makes possible to colorize old ZX-Spectrum applications and games which are not checking the written video data (it's rarely I sure).

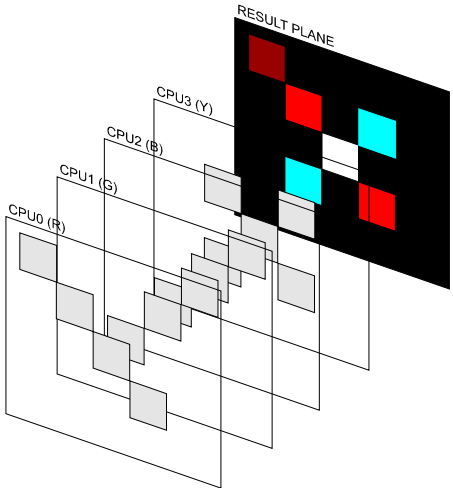


Image 6: Color layers in the videomode 4

Image 7: An instance of the 4th videomode usage

Like the 0-3 video modes, in 4th video-mode every CPU module can use video-memory placed only in either the RAM5 page or the RAM7, it's being selected with the #7FFD port.

The video-mode 5

The video-mode 5 works like the video mode 4 (described above) and it uses the video memory of all processor modules too. The video mode has the 512x384 pixels resolution but the color information is used from attribute memory areas. The mode allows to bring the more high graphic resolution in existing games and utilities (as instance, you can get the Hi-Res version of a text editor if you just correct its loader and fonts as I had made for the ZX-Word). All secrets of the mode in the pixel out technique, **it outs pixels in chess-order!**

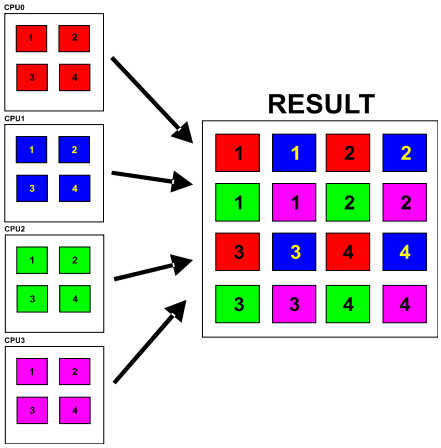


Image 8: The pixel order in the videomode 5

We just take all four screens from 0,1,2,3 modes and their pixels, then we mix the pixels in the chess order and put them on the screen. The mode has the same non-linear spectrum addressing and the same attribute format but the attribute colored pixels are spread on a 16x16 pixel block. As the video source for the video mode, the video controller uses current active video pages (RAM5 or RAM7) of all CPU modules.

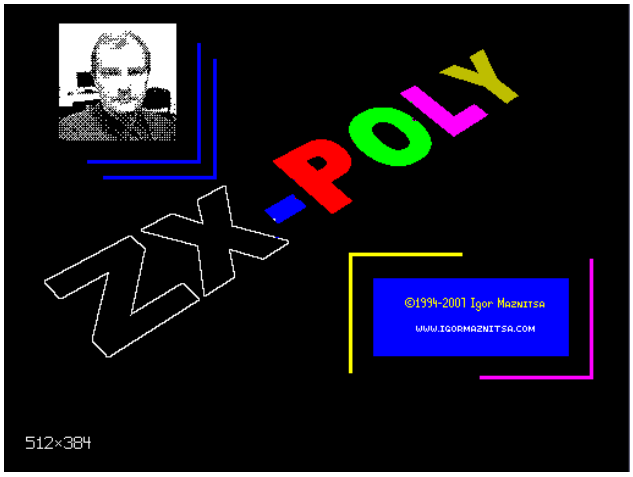


Image 9: An instance of the 5th videomode usage

Synchronization and signals

It plays very important role to make all actions of the processor modules subsequently and there are a few system features for that:

The oscillator

The main base thing – we must (!) use the same oscillator for all our CPU modules!

The system INT,NMI and RESET signals

All system signals must be provided by the same source and the signals must not have their ways through any analogue delay lines because they must reach all CPU modules in the same time!

Keep in your mind that if the #3D00 is not being locked, all global INT and NMI signals are being received only by the CPU0 module but if the the port is locked, those signals will be received by the CPU1-CPU3 modules too. Do not mix system INT, NMI and RESET signals and local module analogues, local signals work by other way. After the local reset for a module (as you read above), the module read the first three bytes from inside and NMI and INT signals will be blocked unless the command byte counter is zero to avoid potential problems.

The WAIT signal

The signal plays very very important role in the system because it is only way to synchronize CPU module works. A CPU module unit will get wait signal if conditions below work:

1. [module_index]<>0 and 0th bit of the #3D00 port is not set.
2. #3D00 is not locked and the module command byte counter == 0 and current_address_bus == the stop-address (ZXPolyR2 << 8 | ZXPoly R1).

We can figure out the current WAIT mode of a processor module through the 2th bit of the ZXPoly R0 module register which shows the pin state.

CPU module units

It is strongly recommended to have all processors in the system of the same manufacturer because it can decrease the spread of their electric characteristics.

How to work with the IO ports

All processor modules work in the same way with IO ports and they can read information from any IO port without any restriction. But the CPU0 module can make any other module as its mapped IO module and all IO operations of CPU0 will change or read the memory spaco of the mapped module. You can ban any processor module through set its ZXPoly R0 4th bit, so its writing IO operations will be ignored.

Possible use cases, as I see them

Old games colorization

It is very important use case for the platform because now we don't have a lot of developers on the platform, so we can get a lot of look good games without their developing from scratch but only through correction. Video modes 4 and 5 give very easy and strong way for that. If you use the video mode 4, you can make those games with 16 colors for every pixel but in the video mode 5 you can radically increase their graphic resolution and use old color schemes.

An algorithm of an old game adaptation (as I see it)

1. We take a game which keeps its sprite information in a non-compressed raw form.

2. With a special utility we recognize areas of the application, which containing the graphic data.
3. Then we make four version of the same memory area, for each color layer of the video mode 4. In each variant we either set or reset the bits on graphic images and the operation makes the image colorful if we see it on a TV. The main possible problem – the game must not check the graphic information which is being out on the screen (!). Unfortunately most advanced games can save their graphic data in a compressed format and it may be will be impossible to correct their graphic data through my method. If the game is active working with attributes for its work time, it is very preferably for us to use the video mode 5 to save important details of game process because we just lost the attribute details in 4th mode (but hi-res is very good too).
4. The whole game with one corrected graphic block should be saved with corrected graphic blocks and a special loader should load all them into CPU modules memory.
5. All start preparing operations for all sections will be made through the memory area of the CPU0 module, because the module has access to the memory of all other modules with the IO module map feature.
6. As the game data had been loaded into memory of all processor modules successfully, the CPU0 returns to its own memory window and write start sequence into inside registers of each module (I think it is a command JP (CALL) <start address>), then sets 4th or 5th video-mode through the #3D00 port, blocks the #3D00 port (to increase compatibility) and send local reset to all modules. The software reset makes all processor modules started with the same memory address simultaneously. If you are going to save game data on a tape or a disk, you have to ban all IO port writing operations for all processor modules exclude the CPU0 module. Also it is possible to load any game data because all modules is working synchronously with the same data from the same IO ports (if they executing the same program!).

A simulation of nonexistent IO devices

1. The CPU0 module loads a device simulating program into the destination CPU module.
2. The CPU0 module writes a simulation start command sequence into the inside simulating module registers to prepare its start.
3. The simulating module is started with a local reset.
4. The CPU0 module maps the simulating module as the IO processing module.
5. If the CPU0 module executes its IO operations, all data will be written not in IO ports but into the simulating module memory area and a local NMI signal will be sent to the module CPU unit (but of course NMI signal can be blocked).
6. If the CPU0 module makes an IO reading, the data will be read from the simulating module memory and the module will be notified through local INT signal that data has been read (you can deactivate it with a DI command).
7. Pay your attention, if the CPU0 has mapped an IO module, it will not be possible for it to work with outside (real) IO ports (exclude the #3D00 and the #7FFD, if the last port is allowed by the module ZXPoly R1 register). The simulating module can use all IO ports and you can use the simulating module as a proxy.

Disk operations

How to use TR-DOS

Because TR-DOS is a very popular Disk OS, it was very important for me to make the platform compatibles with the OS. Every CPU module has its own independent flag of the TR-DOS activation (it is being used to switch on TR-DOS ROM independently for every module) and if the flag is active we will have the listed effects:

1. All ports of a CPU module which are end with #xxxF will be blocked for usage by the module and the TR-DOS ports will be mapped into IO port module space instead. The IO module mapping feature has the highest priority in the system for the CPU0 module and if it is active, the module will work with the memory of the mapped IO module even if the TR-DOS is active, so you can just simulate TR-DOS ports.
2. As in ZX-128, to activate the TR-DOS system you should just execute a command in a #3Dxx address and the 4th bit of the #7FFD port (the ROM1 page) should be set.
3. To deactivate the TR-DOS system you should execute a command in the #4000-#FFFF area or the 4th bit of the #7FFD port should be reset (TR-DOS works only with the ROM1 page).

All CPU modules work with TR-DOS ports in the same way and without and split in module priority so you should be careful when you work with the TR-DOS ports from different processor modules in the same time.

The Afterword

Of course I understand that the platform can generate a lot of questions but keep in your mind that the platform allows to make “impossible” features (for the ZX-128 platform) as “hard but possible” features and that's cool. The main task of the platform is to activate creative processes in developer's minds and give a programmer the possibility to show an “aerobatic” in programming.