ZX Interface Z

Contents

1 Architecture

The *ZX Interface Z* is a ZX Spectrum expansion board. It consists on:

- A dual-core CPU with WiFi and Bluetooth radio
- An Intel Cyclone IV FPGA
- 64MB of RAM, also accessible from the ZX Spectrum
- USB for debugging and programming
- USB host to connect USB devices to the ZX Spectrum

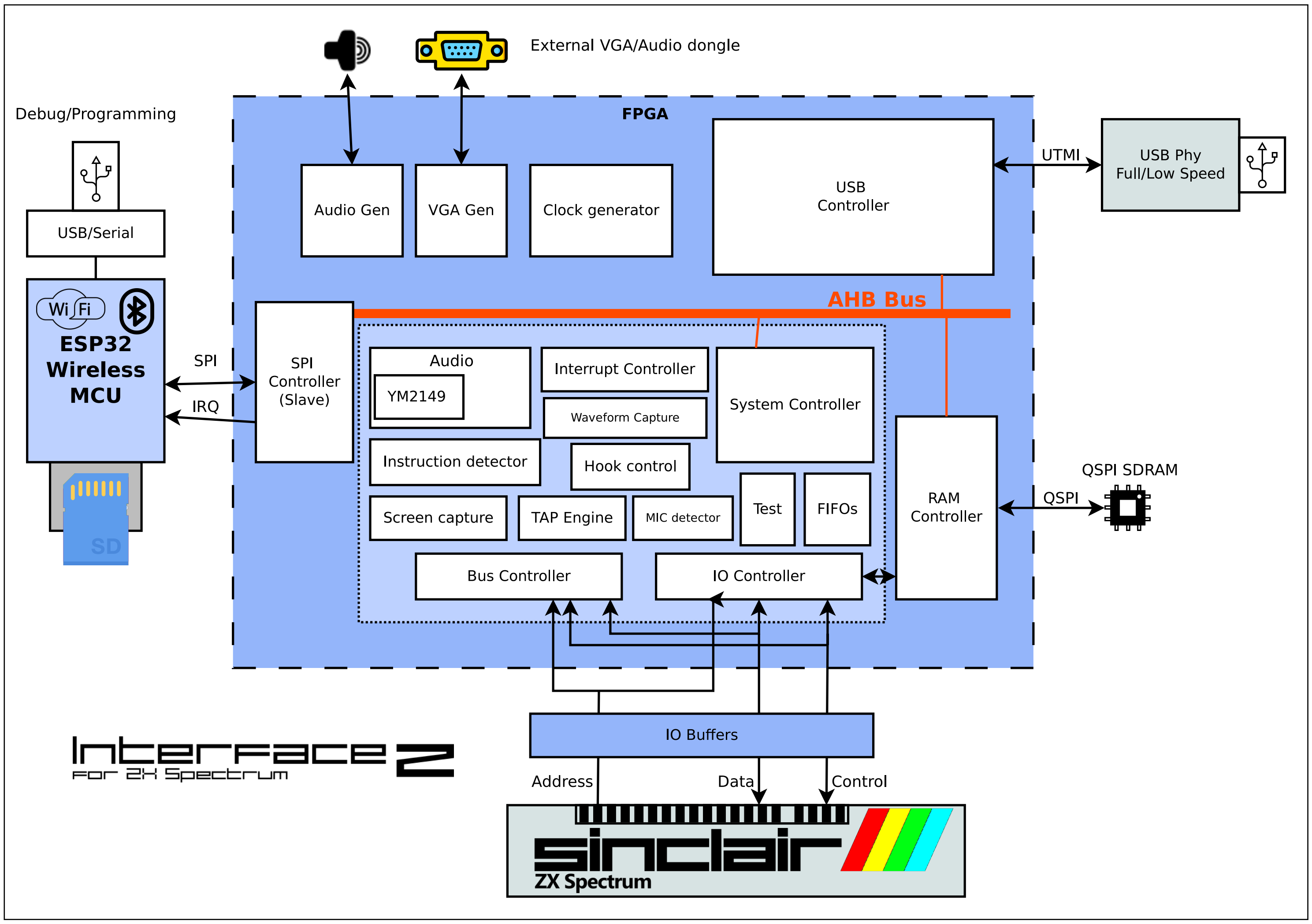

Figure 1: Architecture overview

Architecturally, the board is split into several parts:

- Power supplies
- Spectrum interfacing
- Main system CPU
- Main system FPGA
- USB host
- SDRAM

1.1 Power supplies

The power supplies generate the required voltages for the *ZX Interface Z* to work. Power is drawn from the ZX Spectrum main power supply (either 9V or 12V) and from the 5V ZX Spectrum regulator for USB and some interfacing logic.

1.2 Spectrum interfacing

Interface with the ZX Spectrum is entirely done through the FPGA design.

1.2.1 Data lines

Data lines D0-D7 are connected through bi-directional buffers, whose direction is dictated by the FPGA design.

1.2.2 Address lines

Address lines A0-A15 are unidirectionally buffered from the ZX Spectrum to the FPGA.

1.2.3 /RD, /WR, /MREQ, /IOREQ, /M1, /RFSH, /CK

The main Z80 control lines /RD, /WR, /MREQ, /IOREQ, /M1, /RFSH and /CK are unidirectionally buffered from the ZX Spectrum to the FPGA.

1.2.4 /NMI, /RESET

The /NMI and /RESET lines is driven by the FPGA using open-drain outputs. It is not possible to sample the /NMI or the /RESET line from within the system.

1.2.5 /INT

The /INT line is sampled by the FPGA and can also be driven by the FPGA using an open-drain output

1.2.6 /WAIT

The /WAIT line is driven by the FPGA an open-drain output.

1.2.7 /IORQULA

The /IORQ ULA disable is driven by the FPGA an open-drain output. The functionality is only supported in models which have this line.

1.2.8 /ROMCS, /ROMCS2, /2AROMCS

The /ROMCS, /ROMCS2, /2AROMCS lines are driven by the FPGA open-drain outputs. The functionality depends on the ZX Spectrum model.

1.2.9 /BUSREQ, /BUSACK

The /BUSREQ and /BUSACK are not available on the *ZX Interface Z*

1.3 Main system CPU

The execution core of *ZX Interface Z* operation is performed by an ESP32 system. The ESP32 platform provides an execution environment for the software, WiFi and Bluetooth access. The ESP32 is responsable for:

- Executing the software
- Storing read-only and configuration data
- Programming the FPGA at power up
- Communicating with and receiving events from the FPGA
- Sampling the buttons
- Driving some of the board LEDs
- Providing programming and debugging support via USB serial connection
- Performing analogue sampling of the power lines
- Providing read-write access to the microSD card
- Adding WiFi and bluetooth support

Communication between the ESP32 and the FPGA is done through a dedicated SPI bus and some discrete lines, mostly used for interrupt control.

There is no direct connection between the ESP32 and the ZX Spectrum.

1.4 Main system FPGA

The main FPGA is at the heart of all interaction with the ZX Spectrum and all external devices. The FPGA is responsable for:

- All interaction with the ZX Spectrum
- Providing USB host support
- Capturing and storing video and audio data
- Providing ZX Spectrum devices:
  - Joysticks and mice
  - AY8192 audio implementation
  - ULA “override” - substituting the ULA for certain accesses
- Providing video out capabilities
- Providing audio out capabilities
- Interfacing with the on-board SRAM

1.5 USB host

The USB host function supports USB1.1 Low-Speed and Full-speed.

An UTMI transceiver connects to the USB host controller on the FPGA and to the USB connector proper.

The host controller supports up to 8 simultaneous endpoints, either control, bulk or interrupt. It does not support Isochronous transfers.

1.6 SDRAM

The SDRAM is a quad-SPI SDRAM chip, which is controlled by the FPGA design. It runs at 96Mhz clock, and it's 64Mbit (16MByte) in size. The SDRAM area is logically partitioned according to the purpose.

2 How everything glues together

This section will attempt to explain the basics of how everything glues together, and how the system behaves after powering up.

We will look at the following:

- Start up
- The NMI menu
- The standard ROM hooks

2.1 Start up

When the system starts up (is powered, or after reset) the ESP32 firmware starts. It initialises some of the internal subsystems such as storage on the on-chip flash (this is implemented using a SPIFFS filesystem, whose contents are generated during build of the firmare).

It then loads the FPGA firmware. After the FPGA is ready, a quick test of the external RAM is performed.

Any failure in this sequence will abort further operation and the board LEDs will flash very quickly.

After checking the power supplies for correct voltages, and performing an initial assessment of the ZX Spectrum model, other subsystems are initialised.

The main *ZX Interface Z* ROM is then loaded from the internal storage into memory.

The system then attempts to do a better detection of the ZX Spectrum model. The ZX Spectrum is reset and control is switched to the ZX Interface Z ROM. A quick checksum (partial) of the internal ZX Spectrum ROMs is performed. The checksums are then compared to a list of known checksums. This will be used in the future to disable/enable certain functionalities depending on the ZX Spectrum models.

Once all this is done. the ZX Spectrum is reset to it's internal ROM again, the main ROM hooks are installed and the system is ready to run. At this point all of the required ZX Interface Z subsystems are ready and operational.

2.2 The NMI menu

When user presses the NMI (USR) button, this is detected by the button subsystem and we proceed to the sequence required to show the NMI menu. The sequence is as follows:

- The widgeting system is reset to the NMI menu mode
- The tape players are stopped if they are running
- We request the FPGA to assert the ZX Spectrum /NMI line (which will also include forcing the ROM to the ZX Interface Z ROM).

At this point, nothing else is done on the firmware.

The ZX Spectrum will process the NMI request, and control of the Z80 CPU passes to the ZX Interface Z Spectrum NMI handlerthe ZX Interface Z Spectrum NMI handler routine in nmihandler.asm.

2.2.1 The ZX Interface Z Spectrum NMI handler

Upon entry, the NMI handler saves quite a bit of information to the external RAM (see ZX Spectrum ports and operations). This includes storing all relevant CPU register values, the memory area between Z80 0x4000 and 0x7FFF, the ULA border (as we last saw it) and some other registers like YM2189 mixer. The YM2189 is silenced.

When all information is stored, a command (CMD_NMIREADY) is sent by the ZX Spectrum via the command FIFO.
The NMI handler then waits for main firmware commands on it's main loop

When the main firmware receives the NMI_READY command from Spectrum, it starts building up the menus. For this, first the last framebuffer (which was saved by the NMI handler above) is retrieved to a local framebuffer. All drawing operations are done first on the ESP32 memory, then the whole framebuffer is sent to the external RAM and the ZX Spectrum is notified.

The main windowiwg system loop will check for events from the ZX Spectrum keyboard, and navigate accordingly. When all is done, the relevant commands are sent to the ZX Spectrum for execution.

2.2.1.1 The NMI handler main loop

On the ZX Spectrum, the NMI handler main loop performs the following operations:

- Check for a command from the main firmware. If there is such a command, process it.
- Read the keyboard. If any key was pressed or released, send a CMD_KBDINPUT to the main firmware.
- Check the frame counter in the external memory. If frame counter differs from last frame counter, then the memory area in the external memory from 0x020001 is copied into the ZX Spectrum framebuffer area (at 0x4000), copying both pixel and attribute areas.

If a command is present, then it is executed:

- 0xff: Leave NMI
- 0xfe: Perform a snapshot of the whole memory
- 0xfd: Load a snapshot
- 0xfc: Execute new temporary program
- 0xfb: Read a chunk of RAM memory into the external RAM
- 0xfa: Read a chunk of ROM memory into the external RAM

3 The standard ROM hooks

The *ZX Interface Z* allows for ROM hooks.

A ROM hook consists of an address or a range of addresses on the ZX Spectrum ROM area that whenever an instruction is read from these locations for execution causes the current ROM to switch from the ZX Spectrum ROM into the ZX Interface Z ROM.

This allows the system to “trap” execution of certain ROM routines into our own implementation.

By default, the ZX Interface Z installs two main ROM hooks. One is used for LOAD, and another for SAVE.

Whenever a LOAD or a SAVE is executed the control passes to the ZX Spectrum ROM. The ROM notifies the firmware using the command FIFO of the operation that was requested, and then invokes the NMI handler.

The main firmware, knowing the operation that was requested, invokes the particular menu for that operation.

4 Development for the ZX Interface Z

Development of *ZX Interface Z* is split into five major parts:

- Physical board (PCB) design
- FPGA design
- ESP32 firmware
- Host simulation software
- ZX Spectrum software (ROM)
- Web interface design

4.1 Physical board (PCB) design

The *ZX Interface Z* PCB design has been done using EagleCAD 6.x. The main board uses a four-layer design. The design files can be found in https://github.com/alvieboy/ZXInterfaceZ/tree/master/pcb

The PCB design is considered stable at this point. Changes to the design shall be driven by issues raised in github and accepted.

A series of blog posts have been done to explain in detail how the physical hardware design works, which can be consulted here:

ZX Interface Z - The Hardware, part 1

ZX Interface Z - The Hardware, part 2

ZX Interface Z - The Hardware, part 3

4.2 FPGA design

The FPGA design is almost entirely done in VHDL.

See the FPGA architecture document for details.

4.3 ESP32 firmware

The main *ZX Interface Z* software runs inside an ESP32 chip.

For simplicity, the following subsystems are not described in this document. Refer to the code documentation for more information about these subsystems:

- ADC
- GPIO
- SPI
- Resources
- Built-In Testing
- HDLC encoding/decoding

4.3.1 Startup

This section covers the system start up.

When the system starts up (is powered, or after reset) the ESP32 firmware starts.

All startup is handled inside the main() function.

The startup performs the following initialisations:

- GPIO, console and ADC are initialized.
- The board is initialised.
- The ESP32 LEDs are set to the default values
- The SPI, SDCARD, NVS, WSYS and resource layers are initialised. This includes the internal SPIFFS filesystem.
- The FPGA is programmed from the “intz.bin” present in the SPIFFS and started up
- If FPGA reports BIT mode is requested, BIT mode is entered
- The external RAM is quickly tested
- WiFi is initialized.
- The FPGA firmware is checked for compatibility with the board.
- The interrupt layers is initialised.
- The resources are registered
- The ZX Interface Z ROM is loaded
- Other subsystems are initialised:
  - Configuration
  - Spectrum command handler
  - Video streaming
  - Tape players
  - Device mapper
  - Storage
  - USB
  - Web server
  - VGA
  - Buttons
  - Joystick

After initialization, the spectrum model is further refined by reading the ROM checksums and comparing it to a list of known models.

Finally, the ROM hooks are set up.

The main loop is then entered, which:

- Flashes the LED1
- Restarts the ESP if requested by the firmware update procedure
- Reads the MIC idle status and propagates it.
- Handles serial-over-USB data for the console and the waveform capture
- Processes the buttons

The main loop never exits.

4.3.2 Activity monitor

The activity monitor layer reports activity on the ZX Spectrum MIC line. It is used to time out tape loading and save when working with physical tapes, so that subsequent loads/saves to the external tape do not bring up the tape load/save menu.

Typically, after 10 seconds of inactivity the physical tape is "logically disconnected" and further load/save operations will bring up the load/save menu.

The current mic idle state can be read with the activity_monitor__read_mic_idle() method.

4.3.3 Board detection

Board detection is intimately linked to the model detection.

In boards preceding the r2.3 version, the system lacked ADC support to read the voltage rails. From r2.3 onwards, the voltage rail main supply is connected to the ADC and is used by the board and model detection routines.

A few methods in the board layer can be used to query for the board and supply details.

4.3.4 Model detection

The model detection allows you to understand which ZX Spectrum model is currently hosting the *ZX Interface Z*.

The detecton works at two levels.

The first level is solely based on the board power supply, so it can detect between 9V ZX Spectrums such as the 16K, 48K and 128K gray/toastrack and the +5/+12V spectrums, such as the +2A and +3.

Depending on the model some features might or might not be present.

The most dependant feature is the ULA override (IORQULA) which is not present on +2A or +3, and also not in a few clones like the Investronica 128K.

The method model__supports_ula_override() will inform you if the ULA override method is supported on the model. If the model is not known, it is assumed that it does not support.

Without the ULA override it is not possible to "simulate" a real tape, nor is it possible to inject keyboard presses.

The second level is based on ROM identification. For the ROM identification, a checksum of the two first ROMs is performed but with 64-byte interval between each input. This allows for a quick computation of the checksum.

The checksum is then checked on a list of known model checkusms in order to find out which particular model we are running on.

4.3.5 Buttons

The button layer is responsable for scanning the on-board buttons. It can detect whether a button was pressed, released or long-pressed. Currently the button presses/releases are handled by the process_buttons() in main.c, and the handling is hardcoded. It is planned to make these buttons configurable in the future.

4.3.6 Console

The console layer receives commands from the console (serial-over-USB) and executes them. It's mostly used for debugging purposes.

The console inputs are input into the console layer using console__char() and after a line is received it is processed by console__parse_string(), which maps the main console command to a hander using a lookup table.

The command handlers receive the command as a tokenized list, much like argc/argv style.

In order to add new console commands, create a handler function inside console.c and register it in the table "command_handlers".

4.3.7 Debugger

The debugging infrastructure is still under development.

As of the current version, the debugging infrastructure allows the NMI context to be retrieved for examination. This includes the CPU registers and some other IO registers. Execution code and/or stack contents can be extracted using the memdata subsystem.

The following registers are able to be visualised, and the data corresponds to their value before entering the NMI handler:

- Z80 registers AF, BC, DE, HL, AF', BC', DE', HL', IX, IY, R, I, SP
- ULA register
- YM2189 mixer register

An execution tracer in hardware is planned, but not yet implemented. This infrastructure will allow the user to see exactly which instructions have been executed prior to entering the NMI.

4.3.8 Device Mapper

The purpose of the device mapper is to map device inputs, such as joystick movements, gamepad axis and keys, keyboard presses into concrete actions.

When a specific input changes state, the device mapper is notified (usually by hid__field_entry_char being called from the HID layer), the device mapper checks if the device is mapped to any actions, and if true, what particular action is assigned to that input/field. It then triggers the corresponding action.

Boolean (on/off, such as buttons) and integer fields (such as joystick axis) are supported. The integer fields can be converted into boolean fields by use of low and high thresholds.

The actions that can be configured on a field input are:

- Trigger NMI menu
- Press/release a keyboard key (only models which support ULA override)
- Move or press a button of the Kempston Joystick

- Move or press a button of the Kempston Mouse

Other actions will be added in the future, such as snapshot saving and screen dump.

All the device mapper configurations are stored internally on the “/config” folder in a “devmap.jsn” file. The configurations are stored in JSON format and can be changed in run-time. This folder is not accessible externally.

The following example shows the “devmap.jsn” with two devices. The first device is an USB gamepad, the second device is the *ZX Interface Z* Dual Joystick adaptor, with two interfaces:

```
{
  "devices": [
        {
                "id": "0e8f:0003",
                "manufacturer": "My gamepad",
                "product": "My gamepad",
                "serial": "01298321732",
                "config": "/config/0e8f0003.jsn"
        },
        {
                "id": "0483:5740",
                "manufacturer": "Alvie Systems",
                "product": "Dual Joystick for ZX Interface Z",
                "serial": "01298321732",
                "configs": {
                        "0": "/config/04835740-0.jsn",
                        "1": "/config/04835740-1.jsn"
                }
        }
  ]
}
```

If the device supports multiple interfaces, then Each of the interfaces should point to another JSON file with the particular configuration for that interface. Most devices only support one interface, so only a single config file is required.

On the example above. the gamepad has plenty of buttons and axis. Here is The configuration example for this gamepad:

```
{
        "entries" : [
                { "map": "nmi", "index": 14, "value": 1 },
                { "map": "keyboard", "index": 15, "value": "enter" },
                { "map": "keyboard", "index": 6, "value": "q" },
                { "map": "keyboard", "index": 7, "value": "a" },
                { "map": "keyboard", "index": 4, "value": "o" },
                { "map": "keyboard", "index": 5, "value": "p" },
                { "map": "keyboard", "index": 12, "value": "o" },
                { "map": "keyboard", "index": 13, "value": "p" },
                { "map": "keyboard", "index": 8, "value": "m" },
                { "map": "keyboard", "index": 2, "threshold": 64, "value": "p" },
                { "map": "keyboard", "index": 2, "threshold": -64, "value": "o" },
                { "map": "keyboard", "index": 3, "threshold": 64, "value": "a" },
                { "map": "keyboard", "index": 3, "threshold": -64, "value": "q" }
        ]
}
```

On this mapping, the index "14" which corresponds to a particular button on the gamepad is configured to trigger the NMI menus. Most of the other entries are regular buttons which are mapped to other keyboard keys.

The last four entries are special, and correspond to a joystick axis. The axis "2" is the horizontal axis and the axis "3" is the vertical axis. These axes are known to change between -128 and 127 when the user moves them (they are analog). The threshold selected will convert the axis movement from an analogue point of view into a "on/off" boolean value we need. On this particular example, if axis "2" is greater than 64, then the keyboard "p" key will be pressed (or released if previously pressed and the axis is no longer greater than 64). If axis "2" is less than -64 (more negative), then the keyboard "o" key will be pressed (or released if previously pressed and the axis is no longer less (more negative) than -64).

This is the example configuration for the second interface of the Dual Joystick:

```
{
        "entries" : [
                { "map": "joystick", "index": 0, "threshold": -1, "value": "left" },
                { "map": "joystick", "index": 0, "threshold": 1, "value": "right" },
                { "map": "joystick", "index": 1, "threshold": -1, "value": "up" },
                { "map": "joystick", "index": 1, "threshold": 1, "value": "down" },
                { "map": "joystick", "index": 2, "value": "fire1" },
                { "map": "joystick", "index": 3, "value": "fire2" }
        ]
}
```

This configuration is similar to the previous one, except it will trigger joystick movements, according to the value depicted on the JSON file.

The devmap configuration will be done via the ZX Spectrum. All you need is to move the axis/press the buttons ans configure the action you want.

4.3.9 Disassembler

The *ZX Interface Z* contains a disassembler which can convert a sequence of executable bytes into the textual assembly representation.

The method disassemble__decode() can be used to disassemble an instruction provided the byte sequence and the program counter. It can be called sequentially to decode multiple instructions.

4.3.10 DivMMC compatibility

A minimal layer of DivMMC compatibility is provided at low-level. This should be enabled only for very specific software, such as ESXDOS.

The DivMMC layer implements the standard DivMMC hooks:

- Instruction accesses between 0x3D00 and 0x3DFF
- Instruction accesses between 0x1FF8 and 0x1FFF
- Instruction accesses at restart handlers $00, $08 and $38
- Instruction accesses at load routines 0x04C6 and 0x0562

The DivMMC layer must only be activated after disabling the standard *ZX Interface Z* hooks using rom_hook__disable_defaults() and after loading a DivMMC-aware ROM.

4.3.11 ESXDOS

Support for ESXDOS is planned, but currently not fully implemented.

4.3.12 Slow TAP loading

In certain models it is possible to load TAP and TZX files as if they were regular tapes connected to the EAR input. This allows for all sort of turbo loaders and copy protection schemes to work as they would using a physical tape.

The system "emulates" the tape by capturing accesses to the ZX Spectrum ULA and modifying the EAR value as it would come from a real tape.

Internally the FPGA includes a generic TAP player which is capable of generating the EAR signal from either TAP or TZX file, an injecting it on the ULA output whenever the CPU requests that data.

Since the *ZX Interface Z* requires manipulation of the ULA output to the Z80 it requires the IORQULA line to be connected to the expansion connector of the ZX Spectrum, which is not the case for some models. The method is known to work on the 16K, 48K and 128K (gray) models.

See section Tape player for more details.

4.3.13 Fast tape loading

It is possible to quick/fast load tape files (TAP, TZX and SCR) files by using hooks in the ZX Spectrum ROM. Only standard tapes (which don't use custom loaders like copy protection schemes or turbo loaders) can be loaded using this method.

This works by placing a hook at the LD_MARKER routine in the ZX Spectrum. This routine is responsible for starting the load by detecting the initial sync and then loading the relevant data.

When the Z80 starts executing the LD_MARKER routine, control is transferred to the *ZX Interface Z* ROM. The load hook in the ROM uses SPECTCMD_CMD_FASTLOAD and SPECTCMD_CMD_FASTLOAD_DATA commands to request the main firmware to perform the data load. The main firmware then places the relevant tape content on the external RAM and notifies the *ZX Interface Z* ROM, which copies it to the required location and returns to normal ZX Spectrum operation.

4.3.14 Firmware upgrade

The *ZX Interface Z* allows the firmware to be upgraded using the following methods:

- microUSB cable with dedicated programmer (python)
- microSD card (in development)
- USB memory stick (in development)
- Network (in development)

4.3.15 FPGA interaction

Interfacing with the FPGA is done using a dedicated SPI channel between the ESP32 (master) and the FPGA (slave), and interrupt lines between the FPGA and the ESP32.

All FPGA interfacing is implemented on an API in fpga.c. See documentation present on that file for details of all operations supported.

For more details on the SPI protocol refer to the FPGA design document.

4.3.16 Human Interface Devices

The *ZX Interface Z* supports Human Interface Devices (HID) which comply to the USB Device Class Definition for HID.

Upon retrieving the HID descriptors from the device, those are parsed and eventually mapped in the device mapper.

More details on the HID layer can be found in the implementation.

4.3.17 Joystick handling

The *ZX Interface Z* can respond to Kempston joystick reads on Z80 port 0x1F. The joystick register is 8-bit wide and is implemented on the FPGA.

The contents of the joystick register can be set by modifying the REG_KEMPSTON register on the FPGA. To simplify modification of these values, two methods kempston__set_joystick_raw() and kempston__set_joystick() can be used to change the contents of the kempston joystick.

The kempston joystick should work on all platforms.

4.3.18 Keyboard handling

Keyboard handling (i.e., injecting keyboard presses and releases) on the ZX Spectrum requires ULA override capabilities on the model. Refer to the model documentation to see if your model supports it.

The keyboard data to be injected is implemented on the FPGA on registers REG_KEYB1_DATA and REG_KEYB2_DATA. Only 40 bits are used from this mapping. The two registers are merged together to form a 40-bit value, being REG_KEYB1_DATA used as the lower 32-bits (0-31) and REG_KEYB1_DATA lowermost 8 bits used for the uppermost bits (32-39).

The upper 8 bits of the address line select which part of the 40-bit register is reported. The keyboard bits are reported on the lower 5-bits of the ULA register read. If more than one bit is "reset" on the upper 8-bits of the address lines, then the result is undefined.

Address upper 8-bits	Register bits reported
11111110	REG_KEYB1_DATA(0 to 4)
11111101	REG_KEYB1_DATA(5 to 9)
11111011	REG_KEYB1_DATA(10 to 14)
11110111	REG_KEYB1_DATA(15 to 19)
11101111	REG_KEYB1_DATA(20 to 24)
11011111	REG_KEYB1_DATA(25 to 29)
10111111	REG_KEYB2_DATA(30 to 31)
	REG_KEYB2_DATA(0 to 2)
01111111	REG_KEYB2_DATA(3 to 7)

Injecting keyboard presses is done the following way:

- If the Z80 requests a read from the ULA port, the FPGA will first let the ULA respond, but will interrupt the ULA output just before the Z80 reads the value from it.
- The original read from the ULA is modified to match the injected key presses based on the contents of the address lines which map to the keyboard rows (8 in total).
- The FPGA then outputs the "correct" value to Z80, which is captured in the appopriate register.

Keyboard manipulation is normally done using keyboard__press() and keyboard__release() methods. See documentation of those methods for more details.

4.3.19 LEDs

A total of 6 LEDs are present on the board, and 5 of them are controllable by the system.

One LED is directly driven by the ESP32 system. The other four LEDs are driven by the FPGA.

Currently the LEDs are allocated as follows:

- The upper-left LED (green) to the left of the USR button flashes during normal operation and serves as an indicator of good operation. This LED is under control by the ESP32.
- The LED right to the USR button is controlled by FPGA. Its final purpose is still to be defined.
- The three LED block on the lower-right of the board are controlled by FPGA. Their final purpose is still to be defined.

4.3.20 Reading memory from ZX Spectrum

If the NMI handler is running in the ZX Spectrum it is possible to read memory (either ROM or RAM) from it using dedicated funcions.

However, this process is not entirely trivial due to the following aspects: - The ZX Interface ROM is mapped, so in order to read from original ROM a distinct method must be used - The RAM area between 0x4000 and 0x6000, as it was before the NMI, is stored in the external RAM, so also a distinct method must be used.

To help with this, a request analyser is available with memdata__analyse_request(), which will split a single memory area request into sub-requests that must be executed to fully and coherently read the required area.

This infrastructure is still awaiting improvements.

4.3.21 Pokes

The *ZX Interface Z* supports pokes by using POK files.

A POK file contains tipically several trainers. The POK files can be opened using poke__openfile(). You can iterate through all trainers by using poke__loadentries(). A particular trainer can be applied using poke__apply_trainer().

The poke subsystem uses callbacks for most of the operations. This avoids reading the full POK content into memory. It also allows the subsystem to call a function whose purpose is to ask the user for certain parameters required by the POK trainer. The memory write function is also a callback and can be specialised to the purpose.

4.3.22 Non-volatile storage

The *ZX Interface Z* uses two types of non-volatile storage. Both types will ensure that data stored is available upon next power up or reset.

The special mount "/config" is a SPIFFS filesystem that can be written in run-time. Files stored on this filesystem persist through resets and power cycles. This filesystem does not require any microSD card nor USB storage.

Asides from the "/config" storage, the *ZX Interface Z* supports a simple key-value storage called NVS. This can be used to store basic datatypes as per the following table.

For the data fetch there are two variants of the methods. See implementation for details on the differences between both.

Datatype	Read functions	Write function
uint8_t	nvs__u8/nvs__fetch_u8	nvs__set_u8
uint16_t	nvs__u16/nvs__fetch_u16	nvs__set_u16
uint32_t	nvs__u32/nvs__fetch_u32	nvs__set_u32
float	nvs__float/nvs__fetch_float	nvs__set_float
string	nvs__str/nvs__fetch_str	nvs__set_str

4.3.23 ROM handling

The *ZX Interface Z* supports multiple ROM types. The included ROM for the *ZX Interface Z* is 8KB in size.

It is possible to additionally load a second ROM size by side with the standard one:

```
    - 8KB ROM (with additional 64KB RAM paged in 8KB blocks at 0x2000)
- 16KB ROM
- 32 KB ROM (16KB + 16KB using 128K page registers)
```

4.3.24 ROM hooks

The ROM hooks are places in the ROM area that whenever the Z80 executes instructions from it cause a change in the current mapped ROM.

This is used to implement custom ROM routines without modifying the on-board ROM contents.

The system supports 8 simultaneous ROM hooks. Each hook can map to a specific ROM slot for the 128K and similar models.

Each hook can either be active or inactive.

4.3.24.1 Ranged option

Each ROM hook can be either ranged or not ranged.

4.3.24.1.1 Ranged hooks

A ranged hook changes control of the ROM whenever an instruction is fetched from the range area. The range is specified by a start address and a length field. The ROM mapping is non-persistent, so even if the hook is active and an instruction is fetched from outside the range the system will resume normal operation. This is useful to "patch" small areas of the ROM.

4.3.24.1.2 Non-ranged hooks

A non-ranged hook will modify the mapped ROM when the instruction at the specified address is fetched by the Z80, and the mapping of ROM will persist until a different ROM hook changes it. This is useful for large code blocks where better control of ROM mapping is required.

4.3.24.2 Pre-Post option

Each hook can switch the mapping of the ROM either before or after the first instruction is fetched.

4.3.24.2.1 Pre hooks

For pre-hooks, the control of the ROM changes immediatly when the Z80 attempts to fetch the intruction. It can be seen as if the map changed *prior* to fetching the instruction

4.3.24.2.2 Post hooks

For post-hooks, the control of the ROM changes after the Z80 fetches the first instruction, effectively changing the mapping when the second instruction is executed.

4.3.24.3 Set/Reset option

Each hook can either set the ROM mapping to the internal ROM, or reset the ROM mapping to the ZX Spectrum ROM.

4.3.24.4 Default hooks

A set of default hooks are installed on power up. All these hooks are non-ranged.

Address	Type	Description
0x0767	Pre-Set	LOAD hook. This hook captures execution of the ZX Spectrum LOAD routine
0x04D7	Pre-Set	SAVE hook. This hook captures execution of the ZX Spectrum SAVE routine
0x1FFE	Post-Reset	Return hook. This location has a RET instruction. Jumping to this address causes the ROM control to be switched back to the ZX Spectrum to the address currently in the stack

In order to override a particular ROM routine, the easiest method is to:

- Create the override routine at the same address in the *ZX Interface Z* ROM.
- Add a pre-set hook with the relevant address
- On your override routine, when you wish to go back, place the return address in the stack and then jump into the return hook address. This can be done without modifing any register like in the following example, which will return control to address 0x0123:

```
PUSH    HL              ; Save HL temporarly
LD      HL, $0123       ; Load HL with return address
EX      (SP), HL        ; Swap the stack and HL
JP      $1FFE           ; Jump to the switcher routine
```

4.3.25 Waveform scope

The *ZX Interface Z* supports (in certain firmwares) a scope which is connected to the Z80 signals and to some other internal signals.

The scope supports:

- Up to 64 inputs
- Up to 96MHz sample rate.
- Up to 32 configurable triggers
- A total of 1024 samples

The scope is accessible using the microUSB port (serial-over-USB) using a modified version of the sigrok signal analyser. The Z80 decoder bundled in sigrok can be used to analyse the Z80 lines.

The scope signal types are dynamically loaded from the *ZX Interface Z*, so internal changes in the scope mapping do not require a modification of the sigrok software.

The scope protocol is based in HDLC streaming, and operates simultaneously to the console, so both cannot be used at same time.

4.3.26 SCSI device support

SCSI devices are supported at the SCSI command level. This is used for USB transport protocol SCSI devices, such as USB flash drives.

There are several layers for the SCSI subsystem.

- SCSI command definitions (scsi.h)
- SCSI device support (scsidev.c)
- SCSI disk IO (scsi_diskio.c)

The USB block device driver uses these layers and the internal FAT filesystem and VFS layers to provide access for USB flash drives.

4.3.27 Snapshots

The *ZX Interface Z* supports loading and saving snapshots of the current system state.

The follwing snapshot types are supported:

- SNA snapshots (48K)
- Z80 snapshots (48K)

128K snaphshot types are planned.

Restoration of the snapshots is done by a self-modifying routine which is placed in the ROM. The modification of the ROM values is done using symbol look up tables (see implmentaion in sna_relocs.c for more details).

4.3.28 Spectrum command handling

The ZX Spectrum command handling is perhaps the most important part of the *ZX Interface Z*.

The Spectrum communicates with the rest of the system using the command FIFO channel. This command FIFO channel is used to transfer commands and data, and although it's not the only method, it has some advantages over, for example, exchanging information via the external RAM.

In order to send commands (and some responses) back to the system, the ZX Spectrum performs writes to the command FIFO channel, which resides in the FPGA and is accessible through ports PORT_CMD_FIFO_STATUS and PORT_CMD_FIFO_DATA.

When the command FIFO channel is not empty, the ESP32 system is notified via an interrupt of such fact, and can then read from the command FIFO channel and process the data accordingly.

The commands are structured as a stream of bytes, where the first byte is the command ID, and following bytes, which are variable in size and depend on the command itself. Some commands do not include any more bytes aside from the command ID.

See section ZX Spectrum commands for a list of all available commands.

All command handling is performed in spectcmd.c file.

The command processing from the ESP32 side is as follows:

- Whenever an interrupt notifies us that there is data in the command FIFO channel, the data is placed on a queue and then read from the command processing task. The data is then appended to a command buffer and the buffer is inspected:
- The first byte (command ID) is analysed and a handler for the command is located in the *spectcmd_handlers* array. If such command exists, then the hander is called.
- The command handler analyses the amount of data received so far. If there is not enough data for the command to be executed, the handler simply returns. The macro NEED() can be used to simplify the check for "enough data".
- If enough data exists to execute the command, it is executed and the command data is removed from the command buffer using spectcmd__removedata().

4.3.29 Interrupt handling

Unlike most systems, the interrupt handling on the *ZX Interface Z* is done in two phases.

The only source for interrupts is the FPGA, however several subsystems inside the FPGA can trigger interrupts.

The interrupt method is level driven, so whenever the interrupt line from the FPGA is set the interrupt handler is called.

As with all communication with the FPGA, most interrupt enabling and clearing is done via SPI. But an interrupt can come at any time, and if the SPI channel is busy it cannot be used, so it is not permissible to use the SPI channel from within the ESP32 interrupt handler. But, if the interrupt is not cleared, then the interrupt routine will be called again once it finished execution, leading to an endless interrupt loop. For this reason, a global interrupt acknowledge line is connected between the FPGA and the ESP32.

The behviour of the interrupt lines is as below.

- At reset, the GIEN (Global Interrupt ENable) of the FPGA is active.
- The FPGA, when a subsystem requests an interrupt, if the GIEN is active it activates the interrupt line to the CPU.
- Upon entering the interrupt handler, the CPU quicky asserts the interrupt acknowledge line to the FPGA and stores an event on a queue stating that an interrupt was received.
- The FPGA, detecting an interrupt acknowledge, de-asserts the interrupt line and disables GIEN, therefore disabling any other interrupts.
- An interrupt task in the ESP32 is awken at some point, due to the interrupt queue being not empty. It then reads the FPGA interrupt controller registers via SPI to understand which subsystem triggered the interrupt.
- After checking the interrupt source, the corresponding handler is called to perform the relevant tasks. It is the responsability of this handler to clear the source of the interrupt to prepare for subsequent events.

- When the handler finishes, the ESP32 re-activates the GIEN flag on the FPGA and system resumes the normal operation.

4.3.30 System events

The system events is a publisher/subscriber insfrasructure used to communicate changes in the environment between different subsystems. When a publisher sends an event, all subscribers whose filter matches the event will receive it.

Currently there are five subsystems that can publish events:

- WiFi - Status change event - End-of-scan event
- Network - Network status change event
- USB - USB device status changed event - USB overcurrent event
- Block Device - Block device attach event - Block device detach event
- Storage - Storage attach mountpoint event - Storage detach mountpoint event

System events can be published with systemevent__send(), systemevent__send_with_ctx() and systemevent__send_event(). All events can have an optional context pointer, which is event dependant.

Any subsystem can subscribe to events. The subsystem needs to provide a handler function to be called when events are published. Methods systemevent__register_handler() and systemevent__unregister_handler() can be used to register and unregister handlers, respectively.

New event types can be registered in systemevent.h .

4.3.31 Storage management

The storage management is used to mostly manage the mountpoints as they become available or disappear from the system.

The storage management actively listens for block device events, and preforms mounting and unmounting of filesystems. It also emits changes in the mountpoints using the System events layer.

4.3.32 General Tape Handling

Tape handling is a complex part of the *ZX Interface Z*. The complexity derives from the multitude of options that are available for load and save:

- Load/save from a standard tape
- Load from the emulated tape
- Fast Load/save from/to a TAP
- Fast Load/save from/to a TZX

Every load and save consists of multiple blocks (i.e., “PROGRAM” and “BYTES”, and respective headers). It would be very annyoing if, for every time a block load or block save if requested by the ZX Spectrum, we showed user a menu to choose where to load to/save from. Even for simple games, this would imply popping up the menu 6 or more times just to perform a simple load.

So, in order to simplify the users life and ensure his sanity, a series of timeouts and checks are done. Below, IDLE means that the system had not previously shown any menu to the user (or we entered IDLE on purpose).

- If the system is IDLE, then LOAD/SAVE pops up a menu.
- If the user choose to load or save to/from a standard tape, a 15 second timeout is put in place. If no activity is detected on the physical tape for this amount of time, the system re-enters IDLE mode.
- If the user is loading from an emulated tape, once the emulated tape finishes playing the system re-enters IDLE mode.
- If the user is saving to a emulated tape, then after every saved block a timeout of 4 seconds is used. If no more save requests are made during this timeout, the system re-enters IDLE mode.

4.3.33 Tape player

The tape player resides mostly on the FPGA, and is controlled by the firmware. It consists of a FIFO channel between the ESP32 and the FPGA and a waveform generator on the FPGA. This waveform can then be injected on the ULA register requests.

Two players are implemented in the firmware, one for TAP files and another for TZX files, These implemetations interpret the file contents and send data to the FPGA plater to generate the waveforms.

The FPGA player accepts commands and data using the FIFO channel.

Despite having a single FIFO channel, the FIFO channel accepts commands and data separately.

The FPGA player can generate data pulses, sync pulses and gaps. The T-States associated with those are fully configurable and can even change during a simple block play.

The implemented commands are as follows. The example values given are for a standard TAP file. Note that due to small clock differences between the Z80 T-states and the FPGA T-states, a method is provided (tapeplayer__compute_tstate_delay()) to adjust these values prior to configuring the FPGA player.

- TAP_INTERNALCMD_SET_LOGIC0 (0x80)
  - Set the logic "0" pulse width. For TAP, this is 855 T-States.
- TAP_INTERNALCMD_SET_LOGIC1 (0x81)
  - Set the logic "0" pulse width. For TAP, this is 1710 T-States.
- TAP_INTERNALCMD_GAP (0x82)
  - Play a GAP
- TAP_INTERNALCMD_SET_DATALEN0/1 (0x83/0x84)
  - Set the data length for the following block (i.e., number of bytes that are to be sent). The higher part of this datalen represents the number of bits not used on last data byte.
- TAP_INTERNALCMD_SET_REPEAT (0x85)
  - Repeat the next comand for the specified number of loops
- TAP_INTERNALCMD_PLAY_PULSE (0x86)
  - Play pulse of specified length

- TAP_INTERNALCMD_FLUSH (0x87)
  - Flush contents of the tape.

As a reference, this is how a TAP header player would behave. The following commands/data sequence would play a standard header to the ZX Spectrum (17 bytes):

- SET_LOGIC0 855 /* Pulse "0" width */
- SET_LOGIC1 1710 /* Pulse "1" width */
- SET_DATALEN0 0x0017 /* 17 bytes */
- SET_DATALEN1 0x0000 /* No unused bits in last byte */
- SET_REPEAT 8063 /* Number of pilot repetitions */
- PLAY_PULSE 2168 /* play 8063 pulses of 2168 T-states each */
- SET_REPEAT 0 /* No more repeats */
- PLAY_PULSE 667 /* First sync */
- PLAY_PULSE 735 /* Second sync */
- Load 17 bytes data into the FIFO channel

4.3.34 USB

USB is perhaps one of the most complex parts of the whole *ZX Interface Z* design. This is mostly because the main CPU, an ESP32, does not include any type of USB controller. A USB host controller was developed specifically for this project and is implemented on the FPGA. See the FPGA architecture document for more details about the implementation.

The USB host controller allows communication betweem the ESP32 and USB devices present on the USB bus. The devices can either be connected directly or through a compliant USB HUB (the HUB is itself an USB device).

The USB subsystem is organized in a layered scheme.

At the lowemost level, we have the physical USB controller on the FPGA. The controller has 8 channels, which can be configured independently to perform transactions to a specific end-point of an USB device.

Communication with the physical USB host controller is done via the USB Low Level driver. The low level driver is responsable for handling USB interrupts, configuring channels, submitting transaction requests and transferring data to and from devices.

On top of the USB Low Level layer we have the USB Host layer. The USB host layer creates a logical separation from the USB Low Level and the rest of the firmware, implementing two tasks which are responsable for processing data requests.

On top of the USB Host layer we have the device drivers. The device drivers use the USB Host API to communicate with the USB devices, and use the global API to interface with the system.

At this point there is no direct connection between the ZX Spectrum and the USB layers, except using the drivers indirectly.

4.3.34.1 The Low-level layer

The low level layer is responsable for allocating channels. Prior to any communication a channel needs to be set up for the respective endpoint. See usb_ll__alloc_channel() for details.

4.3.34.2 USB device driver

All USB device drivers need to implement the USB device driver API. The driver API is very simple, and consists of the following methods:

- probe
  - Probe for support for a recently attached device interface
- disconnect
  - Handle disconnection of a device

Upon probing the device, and if the device interface is supported by the driver, the driver shall claim the interface for itself.

4.3.34.3 USB Hub

At a physical level, every USB device is connected to an USB HUB. Internally the USB connector behaves similarly to a HUB with a single port - this is called the root HUB, and it’s implemented in the USB Host interface.

For regular USBHUBs, there are two drivers. One deals with the USB HUB as a HUB device, the other implements the USB HUB logical layer.

4.3.34.4 USB tranfer types

The USB layers supports several type of transfers and requests. These requests can target control, bulk and interrupt endpoints.

4.3.34.4.1 Control requests

Control requests target special control endpoints. All USB devices implenent a control endpoint at endpoint 0.

4.3.34.4.2 Bulk requests

Bulk requests info TODO

4.3.34.4.3 Interrupt

Interrupt requests are asynchronous, and handled by the low-level layers. More info on interrupt requests here TODO

4.3.34.5 USB transfers

Control and bulk requests use a pseudo-synchronous interface, meaning that although all processing is asynchronous by nature, the user interface behaves like a synchronous interface.

Being synchronous means that, whenever a request is placed for completion, the user can wait for it to complete by using one of the completion functions. Additionally the user can set a timeout for the operation to complete.

The main entry point for the USB api is the usbh__submit_request() method.

This method requires the user to fill in a request which is then send to the lower layers for processing. Upon issuing the request, the user can wait for it to complete by using the usbh__wait_completion() method.

The request consist on a structure that should live until the request is finished. In case usbh__wait_completion() is to be called on the same function (or a child) it can reside on the stack.

The following members of the request must be initialised before sending out the request:

- device
  - The device which will receive the request
- target
  - Target memory area for reads and writes.
- length
  - Length, in bytes, of the request. For reads it represents the maximum number of bytes that can be received.
- rptr
  - Read/write pointer. Should be the same as target;
- size_transferred
  - Transferred size. Should be initialised to zero.
- direction
  - Direction for this request, Either REQ_DEVICE_TO_HOST or REQ_HOST_TO_DEVICE.
- control
  - Control flag. Set to '1' if this is a control request.
- retries
  - Number of retries to use for OUT transactions.
- channel
  - Low-level channel to use. Must be allocated first with usb_ll__alloc_channel()

4.3.34.5.1 Control halpers

For control endpoints, a few helper functions exist to simplify the user code:

- usbh__control_msg()
  - Submits a control request and waits for completion
- usbh__get_descriptor()
  - Fetches a descriptor from the device on the endpoint 0

4.3.35 Versioning

The firmware version as well as the ROM version and FPGA version are available for inspection. It is also possible to query the ZX Spectrum model version.

See details in "version.h" for how to extract the versions from the system.

4.3.36 VGA generation

The *ZX Interface Z* includes a VGA generator which supports multiple output modes.

The VGA generation is performed entirely on the FPGA. It is generated from a copy of the ZX Spectrum framebuffer that you usually see between 0x4000 and 0x5AFF. In order to understand what are the contents of the screen all memory transations on the Z80 bus are inspected, and whenever a write is detected, the same write is performed on the framebuffer copy inside the FPGA.

The VGA modes are generated from a base clock of either 46.5Mhz or 28.24Mhz. These frequencies are internally generated by the FPGA PLL. There are plans to use a second dedicated PLL for the VGA which would allow even more modes to be produced.

The following modes are available. The repeat column depicts how many VGA pixels are used for a single ZX Spectrum pixel. The H-border and V-border are the number of VGA pixels used for the border itself.

As of now, the standard PAL mode is not available.

VGA resolution	Aspect	Repeat	H-border	V-border
800x600	4:3	3x	16 pix	12 pix
720x400	16:9	2x	95 pix	8 pix
800x600	4:3	2x	114 pix	108 pix
1024x576 (*)	16:9	2z	256 pix	96 pix

(*) This mode might not be available to use from within the firmware.

4.3.37 Video streamer

The *ZX Interface Z* has a video streamer which sends out the ZX Spectrum framebuffer contents every two frames (25Hz).

The streaming uses a custom payload format and it's sent via UDP packets. The video data is sent out using small fragments which can then be reassembled to display the full image.

4.3.38 Web server

The internal web server is used to provide the web page experience (see [Web Page design]), and to implement a REST interface.

The default port for the web server 80 for the standard buid, and 8000 for the Host simulation software.

4.3.39 WiFi

The WiFi support is a core feature of the *ZX Interface Z*.

The WiFi layer emits some system events which can be captured by the rest of the firmware:

- SYSTEMEVENT_WIFI_STATUS_CHANGED

- Emitted when the WiFi status has changed (connected, disconnected)

- SYSTEMEVENT_NETWORK_STATUS_CHANGED
  - Emitted when the network (TCP/IP) status has changed.
- SYSTEMEVENT_WIFI_SCAN_COMPLETED
  - Emitted when a scan request has successfully completed a network scan

4.3.40 Windowing System bridge

The WSYS bridge links the windowing layer to the rest of the system. It is responsable for dispatching events and implementing the main windowing task. It performs also the bridgding between the "C" and and the "C++" parts of the system.

4.4 Host simulation software

The main firmware is able (to some extent) to be build on a Linux host, using QtSpecem as the ZX Spectrum emulator. The emulator has been slightly modified to acommodate the specific of the *ZX Interface Z*.

4.5 ZX Spectrum software

4.6 ZX Spectrum operations

4.6.1 Command FIFO

The command FIFO is an unidirectional FIFO channel between the ZX Spectrum (sender) and the ZX Interface Z (receiver). When data is written to the command FIFO, the FPGA signals the ESP32 through an interrupt that data is available to be read.

The ZX Spectrum software shall read the PORT_CMD_FIFO_STATUS first to ensure that it can write data to the FIFO. If the FIFO is not full, when data should be written to the PORT_CMD_FIFO_DATA.

4.6.2 Resource FIFO

The resource FIFO is an unidirectional FIFO channel between the ZX Interface Z (sender) and the ZX Spectrum (receiver). The ZX Spectrum software can check for data on the resource FIFO by reading the PORT_RESOURCE_FIFO_STATUS. If data is present, it can be retrieved using the PORT_RESOURCE_FIFO_DATA.

4.6.3 External memory

The external memory is accessible to the ZX Spectrum using IO operations.

In order to access the memory, either read or writes, the memory pointer must be set up first.

The memory pointer is a 24-bit value which represents the external memory address. After every read or write to the external memory this pointer is incremented.

In order to set up the pointer, the ZX Spectrum software must write to the PORT_RAM_ADDR_0, PORT_RAM_ADDR_1 and PORT_RAM_ADDR_2 the value it requires. PORT_RAM_ADDR_0 represent the lower 8-bits, PORT_RAM_ADDR_1 the middle 8-bits and PORT_RAM_ADDR_2 the high 8-bits. The data can then be read or written using the PORT_RAM_DATA port.

As an example, if the software wants to write 0xAA to the external RAM at address 0x55DEAD, then the following sequence should be observed:

- Write 0x55 to PORT_RAM_ADDR_2 port
- Write 0xDE to PORT_RAM_ADDR_1 port
- Write 0xAD to PORT_RAM_ADDR_0 port
- Write 0xAA to PORT_RAM_DATA port

If more bytes are to be read or written sequentially, then there is no need to update the pointer since it increments automatically. For example, in order to write the sequence 0x01, 0x02, 0x03 to the external RAM starting at address 0x55DEAD, then the following sequence should be observed:

- Write 0x55 to PORT_RAM_ADDR_2 port
- Write 0xDE to PORT_RAM_ADDR_1 port
- Write 0xAD to PORT_RAM_ADDR_0 port
- Write 0x01 to PORT_RAM_DATA port
- Write 0x02 to PORT_RAM_DATA port
- Write 0x03 to PORT_RAM_DATA port

4.7 File API

The file API is accessed through the command FIFO. Status responses to the File API should be extracted from the resource FIFO.

The functions implemented by the File API are:

- getcwd
- chdir
- open
- close
- read
- write
- opendir
- readdir
- closedir
- ioctl
- fcntl

4.8 Networking

Networking is avaliable to the ZX Spectrum on a first instance using a mix of command FIFO and resource FIFO reads and writes.

The API is still subject to change.

The networking API is split in two parts. The first part is the low-level API which deal with sockets. The second part is networking utilities, which deal with other upper layer protocols.

4.8.1 Low level networking

The common POSIX API can be used to read and write from network sockets, as if they were files.

The functions implemented by the low-level networking API are:

- gethostbyname
- gethostbyaddr
- socket
- connect
- sendto
- recvfrom

4.8.2 High level networking

The high level networking API provides easy-to-use functions for a set of high level protocols.

The functions implemented by the high-level networking API are:

- wget

4.9 ZX Spectrum ports

The interface ports are partially decoded, and only the lower 8 bits of the address are used.

4.9.1 PORT_SCRATCH0 (R/W)

This port resides at 0x23. It can be used by ZX Spectrum to store and retrieve data temporarly.

4.9.2 PORT_MISCCTRL (R/W)

This port resides at 0x27. It is used for miscelaneous control of the NMI routine.

4.9.3 PORT_CMD_FIFO_STATUS (R)

This port resides at 0x2B, and it's read-only.

When this port is read by the ZX Spectrum it returns the command FIFO status. The command FIFO is an unidirectional FIFO between the ZX Spectrum (sender) and the ZX Interface Z (receiver)

If the returned value is zero, this indicates that the command FIFO can accept new data. If not zero, then the command FIFO is busy and data written to it will probably be lost.

4.9.4 PORT_CMD_FIFO_DATA (W)

This port resides at 0x67 and it's write-only.

When the ZX Spectrum writes to this port, the data will be stored in the command FIFO. The user is responsable for checking that the FIFO is ready before writing new data to it.

4.9.5 PORT_RESOURCE_FIFO_STATUS (R)

This port resides at 0x2F, and it's read-only.

When this port is read by the ZX Spectrum it returns the resource FIFO status. The resource FIFO is an unidirectional FIFO between the ZX Interface Z (sender) and the ZX Spectrum (receiver)

If the returned value is zero, this indicates that the resource FIFO has data.

If not zero, then the resource FIFO is empty.

4.9.6 PORT_RESOURCE_FIFO_DATA (R)

This port resides at 0x33, and it's read-only.

When the ZX Spectrum reads from this port, it will return data stored in the resource FIFO. The user is responsable for checking that the FIFO has data before reading anything from it.

4.9.7 PORT_RAM_ADDR_0 (W)

This port resides at 0x37 and it's write-only. Lower byte for the external RAM pointer.

4.9.8 PORT_RAM_ADDR_1 (W)

This port resides at 0x3B and it's write-only.

Middle byte for the external RAM pointer.

4.9.9 PORT_RAM_ADDR_2 (W)

This port resides at 0x3F and it's write-only.

Higher byte for the external RAM pointer.

4.9.10 PORT_RAM_DATA (RW)

This port resides at 0x63 and it's read-write.

Writing to this port will cause the data to be written to external memory. Reading from this port will return data read from the external memory.

The address used for the external memory is the one set using PORT_RAM_ADDR_0, PORT_RAM_ADDR_1 and PORT_RAM_ADDR_2.

After a read or write, the address is incremented. Successive reads/writes to this port will perform sequencial read/writes to the external RAM.

4.9.11 PORT_MEMSEL (R/W)

This port resides at 0x6b and it's write-only.

Controls the memory page selection of the area between 0x2000 and 0x3FFF. Only the lowermost 3 bits are used.

4.9.12 PORT_NMIREASON (R/W)

This port resides at 0x6f and it's read-write Used to understand why NMI mode has been entered. Currently not used.

4.9.13 PORT_DEBUG (W)

This port resides at 0x7F and it's write-only.

This port is used in host-mode only. Writing to this port will cause the value to be output at the console. This is useful for debugging purposes.

4.10 ZX Spectrum commands

Some commands are deprecated and thus not listed here. In order to send a command, the command ID should be sent first to the PORT_CMD_FIFO_DATA, then the required command arguments shall folow in the format described for each of the commands.

4.10.1 SPECTCMD_CMD_KBDDATA (0x0E)

Send keyboard data to the main system. The arguments to the command are:

- Key data (2 bytes)
  - 2 bytes of data, corresponding to the modifier key first, and the main key after. A value of 0xFF on each represents a key release.

4.10.2 SPECTCMD_CMD_NMIREADY (0x0F)

Notify system that we successfully entered the NMI handler.

This command has no arguments.

4.10.3 SPECTCMD_CMD_LEAVENMI (0x10)

Notify system that we are about to exit the NMI handler.

This command has no arguments.

4.10.4 SPECTCMD_CMD_DETECTSPECTRUM (0x11)

Notify system of the spectrum model.

This command is used at startup, and should not be used elsewhere.

4.10.5 SPECTCMD_CMD_FASTLOAD (0x12)

Notify system that we entered the FAST load routine.

This command has no arguments. This command is pending deprecation.

4.10.6 SPECTCMD_CMD_FASTLOAD_DATA (0x13)

Request fast load data from the spectrum.

The arguments for this command are:

- Block type (1 byte)
  - Represents the spectrum block type
- Block size (2 bytes)
  - 2 bytes, in little-endian format, which depict the size of the block we want to load

The return of this command is placed in the external memory area at address 0x028000. The memory location should be cleared to zero before calling the command. Once the load is complete, a value of 0x01 will be placed at this address. In case of error, a value of 0xff will be placed instead.

4.10.7 SPECTCMD_CMD_ROMCRC (0x15)

Report ROM block for checksum/CRC processing

This command is used at startup, and should not be used elsewhere.

4.10.8 SPECTCMD_CMD_LOADTRAP (0x16)

Notify system we have entered the LOAD trap

4.10.9 SPECTCMD_CMD_SAVETRAP (0x17)

Notify system we have entered the SAVE trap

The arguments for this command are:

- Block size (2 bytes)
  - 2 bytes, in little-endian format, which depict the size of the block we want to save
- Block type (1 byte)
  - Represents the spectrum block type
- Header data
  - 17 bytes of header data, if block being saved is a header

In case a header file is being saved, the header contents (the full 17 bytes) shall be placed in the FIFO, but for other blocks this must be ommited.

The Spectrum shall wait for a response byte in the resource FIFO.

If the response is not 0xFF, then the Spectum shall enter the NMI menu handler.

A second status byte should be fetch from the resource FIFO. If this byte is 0xff, then the save data was cancelled. If the byte was zero, then the save can proceed, and

Once the save is complete,

4.10.10 SPECTCMD_CMD_SAVEDATA (0x18)

Request save data from the spectrum.

Prior to sending this command, the data to be saved shall be placed in the external RAM at address 0x280000. The first byte in the external RAM shall be the block type.

The arguments for this command are:

- Block size (2 bytes)
  - 2 bytes, in little-endian format, which depict the size of the block we want to save

4.10.11 SPECTCMD_CMD_GETCWD (0x20)

TBD

4.10.12 SPECTCMD_CMD_CHDIR (0x21)

TBD

4.10.13 SPECTCMD_CMD_OPEN (0x22)

TBD

4.10.14 SPECTCMD_CMD_CLOSE (0x23)

TBD

4.10.15 SPECTCMD_CMD_READ (0x24)

TBD

4.10.16 SPECTCMD_CMD_WRITE (0x25)

TBD

4.10.17 SPECTCMD_CMD_OPENDIR (0x26)

TBD

4.10.18 SPECTCMD_CMD_READDIR (0x27)

TBD

4.10.19 SPECTCMD_CMD_CLOSEDIR (0x28)

TBD

4.10.20 SPECTCMD_CMD_SOCKET (0x29)

TBD

4.10.21 SPECTCMD_CMD_CONNECT (0x2A)

TBD

4.10.22 SPECTCMD_CMD_SENDTO (0x2B)

TBD

4.10.23 SPECTCMD_CMD_RECVFROM (0x2C)

TBD

4.10.24 SPECTCMD_CMD_GETHOSTBYNAME (0x2D)

TBD

4.10.25 SPECTCMD_CMD_GETHOSTBYADDR (0x2E)

TBD

4.10.26 SPECTCMD_CMD_WGET (0x2F)

4.11 Web interface design

5 Acknowledgements