TIMEX SINCLAIR 2068 Video Board FPGA Design Description

Table of Contents

[1 Introduction 4](#_Toc136616332)

[1.1 Acronyms and Abbreviations 4](#_Toc136616333)

[1.2 Terms and Definitions 4](#_Toc136616334)

[1.3 Referenced Documents 4](#_Toc136616335)

[1.4 General Information 4](#_Toc136616336)

[2 Memory Mapper 5](#_Toc136616337)

[2.1.1 Description 5](#_Toc136616338)

[2.1.2 Z80 Memory Mapper Registers 7](#_Toc136616339)

[2.1.3 Z80 Memory Mapper Registers 8](#_Toc136616340)

[3 FPGA Stack 11](#_Toc136616341)

[3.1 Description 11](#_Toc136616342)

[3.2 Hardware Stack Registers 11](#_Toc136616343)

[4 Video 12](#_Toc136616344)

[4.1 Video Output Select Control 12](#_Toc136616345)

[4.2 TS2068 Video 13](#_Toc136616346)

[4.2.1 Video Mode E11 14](#_Toc136616347)

[4.2.2 ULA Plus and Expanded Mode Video Registers 17](#_Toc136616348)

[4.3 Text Mode Video 21](#_Toc136616349)

[4.3.1 Text Mode Registers 21](#_Toc136616350)

[4.3.2 Text Mode ANSI Codes 23](#_Toc136616351)

[4.4 V9918 Video 25](#_Toc136616352)

[4.4.1 V9918 Registers 26](#_Toc136616353)

[4.5 High Resolution Video 27](#_Toc136616354)

[4.5.1 Hires Memory Spaces and Uses 27](#_Toc136616355)

[4.5.2 Hires Video FSM Commands 28](#_Toc136616356)

[4.5.3 Z80 Access to Hires Video Memory 29](#_Toc136616357)

[4.5.4 Z80 Access to Hires Buffer Memory 31](#_Toc136616358)

[4.5.5 Hires Video Registers 32](#_Toc136616359)

[5 Accessing the ULA Plus Registers 38](#_Toc136616360)

[5.1 Writing ULA Plus Registers 38](#_Toc136616361)

[5.2 Reading ULA Plus Registers 38](#_Toc136616362)

[5.3 ULA Plus Registers 39](#_Toc136616363)

[6 Example Code 43](#_Toc136616364)

[6.1 FPGA Definitions 43](#_Toc136616365)

[6.2 COLOR4 Example 48](#_Toc136616366)

[6.3 COLOR16 Example 53](#_Toc136616367)

# Introduction

## Acronyms and Abbreviations

|  |  |
| --- | --- |
| Acronym or Abbreviation | Description |
| FSM | Finite State Machine  A software or hardware device that performs a specific sequence of operations. The exact sequence is determined by the current state and, commonly, inputs external to the FSM. |
|  |  |
|  |  |
|  |  |
|  |  |

## Terms and Definitions

## Referenced Documents

|  |  |
| --- | --- |
| Document | Location |
|  |  |
|  |  |
|  |  |

## General Information

# Memory Mapper

The memory mapper

### Description

The memory mapper provides access by the TS2068 to the 512k byes of static memory (SRAM) on the video card. The mapper uses 8K chunks just as the native TS2068 does but maps them in in a different manner. This provides 64 chunks of memory in the SRAM.

There are 24 mapping registers configured as three groups of eight. Each group of eight registers is used to map an 8k area of Z80 memory to an 8k area in the external SRAM. The three groups to allow the user to set up different memory maps for Home, EXROM and Dock. The FPGA has been programmed so that any Z80 memory chunk and any TS2068 bank (Home, Exrom or Dock) can be mapped to any of the 64 chunks in the SRAM. This flexible mapping scheme must be used with some caveats. It is possible, and, perhaps, desirable to map out the HOME ROM, video and EXROM in any combination. It is up to the programmer to ensure the memory mapping hardware does not cause conflicts with the computer or any external peripherals when this is done.

Figure 1 shows the algorithm used to map FPGA SRAM into the TS2068 memory space. The hardware monitors the logic level on pin B30 (BUS\_ISO) of the TS2068 connector. If that signal is low, the memory mapper outputs are disabled. This allows other devices on the bus to preempt the memory mapper, preventing it interfering with them.

If the BUS\_ISO signal is high, the memory mapper checks to see which bank is being accessed by the TS2068. It does this by determining whether the HOME, EXROM or DOCK bank is being accessed. If the corresponding bank bit in the mapper enable register is reset, the mapper is disabled and no mapping occurs.

At this point the BUS\_ISO and bank enable bit are high so the mapper can control the bus if applicable. The mapper accesses the mapping value for the chunk in the accessed bank. If bit 7 of the chunk mapping register is set, the mapper will assert the nBE signal and disable the TS2068 memory and place the SRAM memory onto the data bus for Z80 read or write access. If bit 6 of the chunk register is set, the Z80 is allowed to write to the SRAM. Resetting this bit allows the SRAM to act as ROM, if desired.



Figure 1 – Memory mapping flowchart

**Writing/Reading Mapper Registers**

Writing and reading the mapper registers by the TS2068 is fairly easy (see Figure 2). Each of the 24 mapper registers is addressed by first writing its address to the Mapper I/O address register ($B0). Next, the TS2068 writes to or reads from the mapper data register port ($B1). When writing, the value on the Z80 data bus will be written to the desired mapper register while reading will put the data from the register onto the Z80 data bus.

This process must be repeated for each of the mapper registers the programmer wishes to read or write.

**Reading:**

ld a, 7 ; we will be reading from mapper register 7

out ($C6), a

in a, ($C7) ; A now has the value of mapper register 7

**Writing**

ld a, $15 ; chunk 5 of the mapper group 2

out ($C7), a ;

ld a, $22 ; this will map SRAM chunk 5 into EXROM chunk 2 when

; when mapper group 2 is enabled

out ($C7) ; the mapper register is written

**Mapping Memory**

The following steps are required to enable the mapper to control access to the FPGA SRAM.

1. After power-up or reset the memory mapper is disabled because bits 2..0 of the Mapper Map Control register (mapper register $18 (24)) are reset. This prevents any random data in the mapper registers from upsetting the operation of the TS2068. Power on or reset also sets the 24 mapper registers to default values which disable mapping of individual chunks.
2. With the mapper disabled, the programmer must write mapping values to each of the mapping registers they plan to use.   
   Bit 7 must be set to enable that register to map memory.   
   Bit 6 must also be set if write access to that chunk is to be granted.   
    Note that write access for a chunk depends on how that chunk is mapped. For instance, suppose SRAM chunk 15 is mapped into HOME chunk 7 without write access and simultaneously to HOME chunk 4 with write access. The SRAM cannot be written from HOME chunk 7 but can be written from HOME chunk 4. This same condition applies if the same chunk is mapped into different TS2068 banks.
3. Once all of the mapping registers are initialized, select the banks to be mapped by writing to the Mapper Map Control register (register address $18). See the Mapper control register description for details.
4. Any one of the 24 mapping registers can be updated at any time with the proviso that the programmer must ensure that the new mapping scheme does not interfere with the programmer’s code or any support code such as the BASIC ROM.

**Mapper hardware description**

**Mapper registers**

The right side of Figure 2 shows how the mapper registers can be set up to point to the various chunks of FPGA SRAM. Of note is that different mapper registers can point to the same chunk of FPGA SRAM (green) or to no FPGA SRAM (yellow).



Figure 2 – Memory mapping hardware block diagram

### Z80 Memory Mapper Registers

|  |  |  |  |
| --- | --- | --- | --- |
| Port | | Description | |
| C6 | | Register address port | |
| Mode: R/W | | | Reset Value: $00 |
| Bits | | | |
| 7..4 | Unused | | |
| 4..3 | Register bank | | |
| 2..0 | Register number | | |
|  |  | | |
|  |  | | |

|  |  |  |  |
| --- | --- | --- | --- |
| Port | | Description | |
| C7 | | Register data port | |
| Mode: R/W | | | Reset Value: $00 |
| Bits | | | |
| 7..0 | Register value | | |
|  |  | | |
|  | | | |

### Z80 Memory Mapper Registers

#### Mapper Registers $00..$17

|  |  |  |  |
| --- | --- | --- | --- |
|  | | Description | |
| XX | | Register data port | |
| Mode: R/W | | | Reset Value: $00 |
| Bits | | | |
| 7 | ‘1’ allow this chunk to be mapped into TS memory | | |
| 6 | ‘1’ write protect this chunk | | |
| 5..0 | 512k SRAM chunk number | | |
|  | | | |

#### Bank 0

Eight mapper registers controlling the HOME bank

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Reg # | | Description | | | |
| $07..$00 | | Register data port | | | |
| Mode: R/W | | | |  | |
| Register | | | | | |
| Reg # | Chunk | | Bank | | Reset Val |
| $00 | 0 | | Home | | $00 |
| $01 | 1 | | Home | | $01 |
| $02 | 2 | | Home | | $02 |
| $03 | 3 | | Home | | $03 |
| $04 | 4 | | Home | | $04 |
| $05 | 5 | | Home | | $05 |
| $06 | 6 | | Home | | $06 |
| $07 | 7 | | Home | | $07 |
|  | | | | | |

#### Bank 1

Eight mapper registers controlling the EXROM bank

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Reg # | | Description | | | |
| $07..$00 | | Register data port | | | |
| Mode: R/W | | | |  | |
| Register | | | | | |
| Reg # | Chunk | | Bank | | Reset Val |
| $08 | 0 | | EXROM | | $08 |
| $09 | 1 | | EXROM | | $09 |
| $0A | 2 | | EXROM | | $0A |
| $0B | 3 | | EXROM | | $0B |
| $0C | 4 | | EXROM | | $0C |
| $0D | 5 | | EXROM | | $0D |
| $0E | 6 | | EXROM | | $0E |
| $0F | 7 | | EXROM | | $0F |
|  | | | | | |

#### Bank 2

Eight mapper registers controlling the DOCK bank

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Reg # | | Description | | | |
| $07..$00 | | Register data port | | | |
| Mode: R/W | | | |  | |
| Register | | | | | |
| Reg # | Chunk | | Bank | | Reset Val |
| $10 | 0 | | DOCK | | $10 |
| $11 | 1 | | DOCK | | $11 |
| $12 | 2 | | DOCK | | $12 |
| $13 | 3 | | DOCK | | $13 |
| $14 | 4 | | DOCK | | $14 |
| $15 | 5 | | DOCK | | $15 |
| $16 | 6 | | DOCK | | $16 |
| $17 | 7 | | DOCK | | $17 |
|  | | | | | |

#### Mapper Control Register

|  |  |  |  |
| --- | --- | --- | --- |
| Reg # | | Description | |
| $20 | | Mapper Control Register | |
| Mode: R/W | | | Reset Value: $00 |
| Bits | | | |
| 7..3 | unused | | |
| 2 | ‘1’ – enable mapping for DOCK | | |
| 1 | ‘1’ – enable mapping for EXROM | | |
| 0 | ‘1’ – enable mapping for HOME | | |
|  | | | |

# FPGA Stack

## Description

The FPGA stack is a 512 byte LIFO stack similar to the stack in the Z80. The difference is that data is put on and taken off the stack via an I/O port rather than the standard Z80 instructions. This stack may be useful to the programmer as a place to put data that is outside of the normal Z80 memory address space. One example might be to use the stack to store temporary variables when dealing with memory management. Like the Z80 stack, the FPGA stack pointer will wrap around for both reads and writes so the programmer must be vigilant in its use.

The FPGA stack has provision to set the FPGA stack pointer to a user supplied value. Any subsequent writes or reads to the FPGA stack will be performed relative to the new stack pointer value.

## Hardware Stack Registers

|  |  |  |
| --- | --- | --- |
| Addr | Description | |
| C0 | Stack | |
| Mode: R/W | | Reset Value: $00 |
|  | | |
| Writing to this port will push the value onto the stack at the current location and then decrement the FPGA stack pointer  Reading from this port will decrement the FPGA stack pointer and then read the value at the new address | | |

|  |  |  |
| --- | --- | --- |
| Addr | Description | |
| C1 | Stack pointer low byte | |
| Mode: R/W | | Reset Value: $00 |
|  | | |
| Writing to this port will set the low 8 bits (7..0) of the FPGA stack pointer to the supplied value | | |

|  |  |  |
| --- | --- | --- |
| Addr | Description | |
| C2 | Stack pointer high byte | |
| Mode: R/W | | Reset Value: $00 |
|  | | |
| Writing to this port will set the bit 8 of the FPGA stack pointer to the supplied value. | | |

# Video

The FPGA contains video hardware that generates a VGA compatible output that is provided on two standard HD15 connectors. Each of the VGA connectors can display any of the allowed modes and it is possible for both VGA connectors to display the same mode. In this document we will label the VGA connectors as VGA\_A (P7) and VGA\_B (P8). Each of the video generators (Timex, Text mode, Hires and TMS9918) has its own dedicated memory, so they do not interfere with one another.

## Video Output Select Control

The selected video mode for each of the VGA connectors is controlled via dedicated registers accessed via ports $BF3B and $FF3B. See section xxxx for instructions on reading and writing the registers using these ports.

|  |  |
| --- | --- |
| Reg # | Function |
| 142/$87 | Selects the video mode for VGA\_A |
| 143/$88 | Selects the video mode for VGA\_B |

|  |  |
| --- | --- |
| Reg Value (binary) | Function |
| 10xx xxxx | Select 80 column text for output |
| 01xx xxxx | Select VDP9918 core for output |
| 00xx xx00 | Select Timex/ULA Plus video for output |
| 00xx xx01 | Select Hires for output |
| 00xx xx10 | Timex video in foreground, hires video in background |
| 00xx xx11 | Hires video in foreground, Timex video in background |
| Note: ‘x’ indicates a don’t care bit value | |

Example code:

ld bc, $BF3B ; select VGA\_A mode register

ld a, $8E

out (c), a

ld b, $FF ; set VGA\_A for 80 column text

ld a, $80

out (c), a

ld b, $BF ; select VGA\_B mode register

ld a, $8F

out (c), a

ld b, $FF ; set VGA\_B for Hires video

ld a, $01

out (c), a

## TS2068 Video

The Timex video modes allow all the standard Timex video modes as well as some expanded modes that provide enhanced capabilities. Accessing these capabilities is a bit complicated by the need to maintain compatibility with legacy software.

The following modes are available:

|  |  |
| --- | --- |
| Mode | Description |
| S0 | This is the standard video mode that is active when the TS2068 boots. The video bit map and attributes are as defined in the TS2068 manual and reside in display file zero ($4000..$5FFF). This mode allows up to 48k of contiguous RAM in the home bank |
| S1 | This is an alternate video mode that can be activated. The video bit map and attributes are as defined in the TS2068 manual and reside in display file zero ($6000..$7FFF). The system variables and RAM resident routines are moved to high memory. It is expected that both display files will be in use for enhanced video output. This mode allows for up to 32k of contiguous RAM in the home bank. A total of 48k is available if display file 0 is not used however, since BASIC requires contiguous memory, the memory in display file 0 cannot be used for BASIC code. |
| S2 | In this mode, the TS2068 uses display file 0 for the display bit map and display file 1 for the attributes. This allows each scan line to have 32 characters with unique foreground/background colors. |
| S6 | In this mode, the TS2068 reads alternate bit maps from display file 0 (character positions 0, 2, 4…) and display file 1 (character positions 1, 3, 5…). The attribute byte is read from bits 5..3 of the register at port $FF.  This mode allows for 64 column (or 80 column with redefined characters) character display or a 512x192 bit mapped display. |
|  | |
| E11 | This mode uses both display files to provide a sixteen (16) color bit map mode. The screen resolution is 128 pixels wide by 192 pixels tall. Each byte of display memory supplies color data for two contiguous pixels. The screen is split in half horizontally with the top 96 scan lines supplied by DF0 and the lower 96 scan lines supplied by DF1. The color for each pixel is selected from the ULA Plus palette registers. |
| E14 | This mode uses both the bit map and attributes in both display files. It is like mode S6 but the attributes are used instead of the fixed attribute as in standard S6 mode. |
| E15 | This mode uses both display files to provide a four (4) color bit map mode. The screen resolution is 256 pixels wide by 192 pixels tall. Each byte of display memory supplies color data for four contiguous pixels. The screen is split in half horizontally with the top 96 scan lines supplied by DF0 and the lower 96 scan lines supplied by DF1. The color for each pixel is selected from the ULA Plus palette registers. |

This document will not describe the standard video modes in detail since they are described in the TS2068 user’s manual and the TS2068 technical manual.

### Video Mode E11

Video mode E11 provides a 128X192 @16 color per pixel display mode sacrificing horizontal resolution for color depth. Using mode E11 with the BASIC ROM routines requires placing the TS2068 into the 64 column mode by invoking the video change service routine. This moves the machine stack and RAM resident code to upper memory. If the programmer takes complete control of the machine, the video change service routine need not be run but the programmer must then provide all required services including any keyboard interrupt routines.

The programmer must also load the ULA palette registers with the appropriate 8 bit gggrrbb color value for each of the 16 allowed colors.

Mode E11 reads data from DF0 to populate the upper half (top 96 scan lines) of the screen and DF1 for the lower half of the display. This is required to maintain compatibility with TS BASIC. The memory is read sequentially and so does not follow the video memory scheme used by the standard TS2068 video modes. The figure below shows the mapping of the pixel number versus the video memory address. Each video address supplies color information for two pixels with bits 7..4 providing the color for the even numbered pixel while address 3..0 provides the color for odd numbered pixels. For instance, writing $37 to address $4001 would specify color #3 to pixel #2 and color #7 to pixel #3.



Pixel Number vs Memory Address For 128x192 16 color mode

### Video Mode E15

Video mode E15 provides a 256X192 @4 color per pixel display mode. Using mode E15 with the BASIC ROM routines requires placing the TS2068 into the 64 column mode by invoking the video change service routine. This moves the machine stack and RAM resident code to upper memory. If the programmer takes complete control of the machine, the video change service routine need not be run but the programmer must then provide all required services including any keyboard interrupt routines.

The programmer must also load the ULA palette registers with the appropriate 8 bit gggrrbb color value for each of the 4 allowed colors.

Mode E15 reads data from DF0 to populate the upper half (top 96 scan lines) of the screen and DF1 for the lower half of the display. This is required to maintain compatibility with TS BASIC. The memory is read sequentially and so does not follow the video memory scheme used by the standard TS2068 video modes. The figure below shows the mapping of the pixel number versus the video memory address. Each video address supplies color information for two pixels with bits 7..4 providing the color for the even numbered pixel while address 3..0 provides the color for odd numbered pixels. For instance, writing $37 to address $4001 would specify color #0 to pixel #4, color #3 to pixel #5, color #1 to pixel #6 and color #3 to pixel #7.



Pixel Number vs Memory Address For 256x192 4 color mode

### Video Overlay Modes

The video hardware provides an overlay mode that can be specified for one or both video outputs. The overlay modes allow the Timex and hires video outputs to be displayed simultaneously. This is done by specifying one or two transparent colors and then specifying either the Timex or hires to be the foreground layer. When this is done, the video hardware outputs the foreground video until it sees one of the specified transparent color pixels. When a transparent pixel if found, the background pixel is displayed. This allows the programmer to set up rich displays that utilize both the Timex and hires hardware to best effect.

Two transparent colors are provided to provide some flexibility in specifying transparency. Both transparent colors are always checked, so, if only one transparent color is desired, the programmer must set both registers to the same color value.

### ULA Plus and Expanded Mode Video Registers

#### ULA I/O Port Addresses

|  |  |  |
| --- | --- | --- |
| Addr | Description | |
| BF3B | Expanded Video Mode register address | |
| Mode: RW | | Reset Value: $00 |
| Bits | | |
| 7..6 | Register group  00 palette registers  01 palette control  10 expanded mode registers  11 unused | |
| 5..0 | Register address | |

|  |  |  |
| --- | --- | --- |
| Addr | Description | |
| FF3B | Expanded Video Mode data address | |
| Mode: RW | | Reset Value: $00 |
| Bits | | |
| 7..0 | See specific registers below | |

#### ULA Register Description

##### Palette Registers

These 64 registers hold the 8 bit color values (gggrrrbb) for the ULA Plus palettes. They are also used to hold the color values for the hires graphics mode.

|  |  |  |
| --- | --- | --- |
| Addr | Description | |
| BF3B:00..3F | Palette Registers | |
| Mode: RW | | Reset Value: $00 |
| Bits | | |
| 7..0 | Palette color value  GGGRRRBB | |

##### ULA Plus Enable Register

This register enables/disables the ULA Plus mode for modes S0, S1, S2 and S6.

|  |  |  |
| --- | --- | --- |
| Addr | Description | |
| BF3B:40 | ULA Plus Control | |
| Mode: RW | | Reset Value: $00 |
| Bits | | |
| 7..1 | unused | |
| 0 | 1 enable ULA Plus mode  0 disable ULA Plus mode | |

##### Expanded Video Mode Control

This register holds the control values for the expanded video modes. If bit 7 is reset the values in the TS2068 registers $FF and $FE are used to control the video mode. If bit 7 is set, the values in this register are used to set the video mode. All of the ‘Ex’ video modes must be enabled via this register. For instance, to select mode E11 (64 column text with full attributes, a value of $8B (139) must be written to this register.

The standard video modes can be selected using this register. For example, to select mode S6 (64 column text with fixed ink/paper values) one would write the binary value 1xxx0110 to this register where xxx sets the ink/paper combination. This will configure the FPGA hardware to output 64 column mode that is the same as the standard TS2068 64 column video mode.

|  |  |  |
| --- | --- | --- |
| Addr | Description | |
| BF3B:80 | Expanded Mode Control | |
| Mode: RW | | Reset Value: $00 |
| Bits | | |
| 7 | 1 enable expanded video mode  0 disable expanded video mode | |
| 6..4 | Selects the ink/paper colors for the 64 column text mode S6 | |
| 3..0 | 0 select mode S0  1 select mode S1  2 select mode S2  6 select mode S6  11 select mode E11  14 select mode E14  15 select mode E15 | |

##### Unused Registers

These two registers are unused.

|  |  |  |
| --- | --- | --- |
| Addr | Description | |
| BF3B:81 | Unused Register | |
| Mode: RW | | Reset Value: $00 |
| Bits | | |
| 7..0 |  | |

|  |  |  |
| --- | --- | --- |
| Addr | Description | |
| BF3B:82 | Unused Register | |
| Mode: RW | | Reset Value: $00 |
| Bits | | |
| 7..0 | unused | |

##### Expanded Border Color

|  |  |  |
| --- | --- | --- |
| Addr | Description | |
| BF3B:83 | Expanded Border Color | |
| Mode: RW | | Reset Value: $00 |
| Bits | | |
| 7..0 | Color value (gggrrrbb) for the border in expanded video mode | |

##### Spectrum Black Control

This register controls whether there is one or two black levels

|  |  |  |
| --- | --- | --- |
| Addr | Description | |
| BF3B:84 | Spectrum Black Control | |
| Mode: RW | | Reset Value: $00 |
| Bits | | |
| 7..1 | Unused | |
| 1 | ‘1’ – use only one shade of black as in the ZX Spectrum | |

##### Hires Color Bank

This register allows selection of the section of the ULA register set used in 4 and 16 color modes.

In 4 color mode there are 16 possible banks of color while there are 4 banks in 16 color mode.

|  |  |  |
| --- | --- | --- |
| Addr | Description | |
| BF3B:85 | Color Bank | |
| Mode: RW | | Reset Value: $00 |
| Bits | | |
| 7..4 | unused | |
| 3..0 | Border color for expanded ULA Plus mode | |

##### Transparent Color #1

This register sets the first of two transparent colors

|  |  |  |
| --- | --- | --- |
| Addr | Description | |
| BF3B:86 | Transparent Color #1 | |
| Mode: RW | | Reset Value: $00 |
| Bits | | |
| 7..0 | Color value (gggrrrbb) for the first transparent color | |

##### Video Output A Source select

This register selects which of the video sources will be displayed on output A

|  |  |  |
| --- | --- | --- |
| Addr | Description | |
| BF3B:87 | Video A Output Select | |
| Mode: RW | | Reset Value: $00 |
| Bits | | |
| 7..0 | 10xxxxxxb – select the text mode terminal  01xxxxxxb – select the V9918 output  00xxxx00b – select the Timex video  00xxxx01b – select the hires output  00xxxx10b – Timex foreground and hires background  00xxxx11b – Hires foreground and Timex background | |

##### Video Output B Source select

This register selects which of the video sources will be displayed on output B

|  |  |  |
| --- | --- | --- |
| Addr | Description | |
| BF3B:88 | Video B Output Select | |
| Mode: RW | | Reset Value: $00 |
| Bits | | |
| 7..2 | unused | |
| 1..0 | Bank number | |

##### Transparent Color #2

This register sets the second of two transparent colors

|  |  |  |
| --- | --- | --- |
| Addr | Description | |
| BF3B:89 | Transparent Color #2 | |
| Mode: RW | | Reset Value: $00 |
| Bits | | |
| 7..0 | Color value (gggrrrbb) for the second transparent color | |

## Text Mode Video

The text mode video is a simplified version of Grant Searle’s video terminal for his Multicomp computer system. The terminal is simplified by omitting any of the keyboard functions and reworking the status port bit assignments.

The video output is an 80x25 character display with color attributes. Control is accomplished by using ANSI escape sequences as noted on Grant’s website (<http://searle.x10host.com/Multicomp/index.html>).

To print a character, simply output the character to port $A2 and the terminal will do the rest. Writing a character to the display can generally be done at the full Z80 speed unless scrolling of the display is required. This scrolling is automatic and can cause characters to be missed if this is not kept in mind. Therefore, it is important to check the status bits to determine whether the terminal is ready for another character. Any ANSI escape sequence such as scrolling or deleting multiple characters will also likely take enough time that testing the busy bit should be done.

### Text Mode Registers

|  |  |  |
| --- | --- | --- |
| Addr | Description | |
| A2 | Character port for text mode video display | |
| Mode: W | | Reset Value: $00 |
| Bits | | |
| 7..0 | Character to display | |

|  |  |  |
| --- | --- | --- |
| Addr | Description | |
| A2 | Status Port for text mode video display | |
| Mode: R | |  |
| Bits | | |
| 7 | 1 – interrupt (not useful) | |
| 6 | 1 - the display is busy. This will be asserted if the operation triggered by an escape sequence is busy. It will also be asserted when scrolling the display.  It is recommended that this bit always be used to determine if the display is ready for another character. | |
| 5 | Always 0 | |
| 4 | Always 0 | |
| 3 | Always 0 | |
| 2 | Always 0 | |
| 1 | 1 when the character has been written to the display | |
| 0 | Always 0 | |

### Text Mode ANSI Codes

**Supported ANSI Codes**

The TV/monitor display supports the following "escape" sequences

| SEQUENCE | NAME | DESCRIPTION | EXAMPLE |
| --- | --- | --- | --- |
| ESC[H | Cursor Home | Moves the display cursor to top-left | PRINT CHR$(27);"[H"; |
| ESC[K | Erase EOL | All characters on current line from cursor position to end replaced with spaces | PRINT CHR$(27);"[K"; |
| ESC[s | Save cursor pos | Current row and column saved | PRINT CHR$(27);"[s"; |
| ESC[u | Restore cursor pos | Return cursor to previously stored row and column | PRINT CHR$(27);"[u"; |
| ESC[2J | Clear screen | Same as PRINT CHR$(12) - clear screen and move cursor to top left | PRINT CHR$(27);"[2J"; |
| ESC[0J or ESC[J | Clear to end of screen | All positions from the current cursor to end of screen are replaced with spaces | PRINT CHR$(27);"[0J"; |
| ESC[{val}A | Cursor up | Move cursor up {val} places | PRINT CHR$(27);"[3A"; |
| ESC[{val}B | Cursor down | Move cursor down {val} places | PRINT CHR$(27);"[2B"; |
| ESC[{val}C | Cursor forward | Move cursor right {val} places | PRINT CHR$(27);"[15C"; |
| ESC[{val}D | Cursor backward | Move cursor left {val} places | PRINT CHR$(27);"[1D"; |
| ESC[{row};{col}H | Set cursor position | Move the cursor to row {row} and column {col} (1;1=top left) | PRINT CHR$(27);"[5;33H"; |
| ESC[L *Only ONE line supported* | Delete line | Delete the current cursor line and move the display underneath it up one line. Last line of the display then is blank. | PRINT CHR$(27);"[L" |
| ESC[M *Only ONE line supported* | Insert line | Insert a blank line at the current cursor position and move the display below it down one line. | PRINT CHR$(27);"[M" |
| ESC[{val}m ESC[{val1};... {val8}m  *(1 to 4 values)* | Set graphics rendition  Note: Text and background setting reversed if inverse is active. | Set the active foreground or background text colour. If character attributes enabled, all characters that appear after this sequence will use the colour specified. If character attributes disabled then the complete screen will be displayed in the colours specified.  One or two parameters can be specified in the same command.   |  |  | | --- | --- | | val | effect | | 0 | Reset atts to default | | 1 | Bold (bright) text on | | 7 | Inverse colours (back<=>text) | | 22 | Bold (bright) text off | | 27 | Normal colours | | 30 | Text BLACK | | 31 | Text RED | | 32 | Text GREEN | | 33 | Text YELLOW | | 34 | Text BLUE | | 35 | Text MAGENTA | | 36 | Text CYAN | | 37 | Text WHITE | | 40 | Background BLACK | | 41 | Background RED | | 42 | Background GREEN | | 43 | Background YELLOW | | 44 | Background BLUE | | 45 | Background MAGENTA | | 46 | Background CYAN | | 47 | Background WHITE | | 90 | Text GREY | | 91 | Text BRIGHT RED | | 92 | Text BRIGHT GREEN | | 93 | Text BRIGHT YELLOW | | 94 | Text BRIGHT BLUE | | 95 | Text BRIGHT MAGENTA | | 96 | Text BRIGHT CYAN | | 97 | Text BRIGHT WHITE | | 100 | Background GREY | | 101 | Background BRIGHT RED | | 102 | Background BRIGHT GREEN | | 103 | Background BRIGHT YELLOW | | 104 | Background BRIGHT BLUE | | 105 | Background BRIGHT MAGENTA | | 106 | Background BRIGHTCYAN | | 107 | Background WHITE | | PRINT CHR$(27);"[31;44mTHIS IS RED ON BLUE" PRINT CHR$(27);"[93;42mTHIS IS BRIGHT YELLOW ON GREEN" PRINT CHR$(27);"[30mTHIS IS BLACK TEXT" PRINT CHR$(27);"[41mBACKGROUND NOW RED" will give the following...  THIS IS RED ON BLUE THIS IS BRIGHT YELLOW ON GREEN THIS IS BLACK TEXT BACKGROUND NOW RED |

## V9918 Video

**As of 12 June 2023 the V9918 core is not working**

The TMS9918 core is programmed as described in the Texas Instruments data sheet. The output is selected by following the instructions in section 4.1

### V9918 Registers

|  |  |  |
| --- | --- | --- |
| Addr | Description | |
| A0 | VDP register port | |
| Mode: W | | Reset Value: $00 |
| Bits | | |
| 7..0 | Character to display | |

|  |  |  |
| --- | --- | --- |
| Addr | Description | |
| A1 | VDP memory read/write port | |
| Mode: W | | Reset Value: $00 |
| Bits | | |
| 7..0 | Character to display | |

## High Resolution Video

The high resolution video mode provides 256x192 pixels at 256 colors per pixel. The colors are mapped in the byte as gggrrrbb. The hires mode is controlled by accessing registers via ports $BF3B and $FF3B. See section for instructions on reading and writing the registers using these ports.

The hires subsystem is designed access its memory based on a rectangle method. The programmer defines a rectangle by writing to the appropriate registers and then any reads, writes, or copies are performed on that rectangle. Rectangles can be as small as one pixel or as large as the whole screen.

The border color for any mode using the hires screen will come from the HR\_BORDER register

### Hires Memory Spaces and Uses

#### Hires Video Memory

The hires video memory contains the color values for each pixel of the 256x192 hires window. This 48kBi memory is implemented as true dual port static RAM which allows the Z80 to access the memory via the control FSM on one port. The second port is used by the video display circuity to output the picture information for display on the monitor. Using a true dual port static RAM allows simultaneous access by the control FSM and the video output controller resulting in no wait states for either function.

#### Hires Buffer Memory

The hires Buffer memory is a 4kBi memory configured as 256x16 memory. This memory is available for the programmer’s exclusive use. The intent of the memory is to provide a scratchpad for fast copying between the buffer memory and video memory in either direction. The buffer memory acts much the same the video memory by responding to copy instructions as well as the auto-increment feature of memory writes. For instance, the programmer may write rectangular tiles of any size into buffer memory and then use the hires video FSM to quickly transfer them to the video screen. Another option would be to temporarily hold video screen data by transferring it to the buffer, performing some operation on the video memory, and then transferring the data from the buffer to the same or a different screen location.

The programmer is cautioned that there is only 4kBi of memory, so use it wisely.

### Hires Video FSM Commands

The hires subsystem uses a FSM to control access by the Z80 to the hires video memories. A direct connection is undesirable because dual access would be complex and is unnecessary. The FSM is clocked at 100MHz so it is fast enough that standard Z80 memory reads and writes via Z80 I/O instructions can be accommodated without wait states.

The FSM provides hardware acceleration for common functions via writing to the command register located at $A1 in the ULA Plus register area.

#### NOP – No operation ($00)

When this command is written to the command register, nothing is done. This command is primarily used to place the FSM in a known state.

#### FILL – fill a rectangular region ($01)

This command fills a rectangular region with a specified color value. Before invoking this command, the rectangular region must be specified using the ULX, ULY, LRX and LRY registers. The fill color must also be specified in the HR\_FILL register.

If an operation code has been written to the HR\_OP register, the value written to the video memory will be the value obtained by performing the operation on the value in HR\_FILL and the value already in that memory location. For instance, if the HR\_OP value is HR\_XOR, the value written will be the XOR of the fill value and the value currently at the memory location to be written.

The command may take a significant amount of time so it is highly recommended reading the HR\_CMD register and testing the “busy bit” which is bit 7. This bit will be set if the FSM is busy.

#### COPY – copy a rectangular region ($02)

This command copies a rectangular region from one location to another location. Before invoking this command, ULX, ULY, LRX and LRY must be set to the location of the rectangle to be moved. Registers DESTX and DESTY must be set to the desired upper left corner of the destination location. This command properly handles copying from any location to any location in video memory.

If an operation code has been written to the HR\_OP register, the value written to the video memory will be the value obtained by performing the operation on the value in the destination memory location and the value already in the destination memory location. For instance, if the HR\_OP value is HR\_XOR, the value written will be the XOR of the source memory location and the destination memory location.

#### BTOS – copy a rectangle from the buffer memory to the screen memory. ($03)

This command copies a rectangle in the hires buffer memory to a destination in screen memory. The normal HR\_OP rules apply. Before invoking this command, ULX, ULY, LRX and LRY must point to the rectangle in buffer memory. Note that the maximum value of ULY and/or LRY is 15. See the section describing the buffer memory. DESTX and DESTY must point to the upper left corner of the destination in the screen memory.

#### STOB – copy a rectangle from the screen to buffer memory. ($04)

This command copies a rectangle from the screen memory to a destination in buffer memory. HR\_OP rules do NOT apply. Before invoking this command, ULX, ULY, LRX and LRY must point to the rectangle in screen memory. DESTX and DESTY must point to the upper left corner of the destination in the buffer memory. Note that the maximum value of DESTY will be 15. See the section describing the buffer memory.

### Z80 Access to Hires Video Memory

#### Non-auto Increment

This access mode allows reading and writing to the hires memory at any random pixel location. This mode is probably most useful when the programmer wishes to perform read-modify-write operations on only a few bytes. This method is slow since the address of the desired pixel must be updated whenever a new pixel location is desired.

The programmer can access the hires memory a byte at a time for reading or writing by using the following method:

1. Write any value to the HR\_OP register with bit 7 set ($80 will do). This step disables the auto increment mode.
2. Write the ULX, ULY registers with the desired pixel location. Writing either or both of ULX or ULY will initialize the hires memory address register and MUST be done if a different hires memory address is required.
3. Write $A0 to the ULA address port at Z80 I/O address $BF3B.
4. Read or write the hires byte as desired by accessing the ULA data port at Z80 I/O address $FF3B.
5. Repeat step 2 to 4 until all desired data is read/written.

#### Auto Increment

This method allows the programmer to easily and quickly read rectangular blocks of memory without the need to compute memory address for each pixel. Z80 access to hires memory is accomplished by reading or writing to ULA port $A0. Access to the hires memory in this mode is based on a user defined bounding rectangle. After the hires registers have been properly configured, reads or writes to ULA port $A0 will sequentially read/write byes in the hires memory. The reads/writes move along a line and then wrap to the start of the next line and continue thusly until the end of the bounding rectangle is reached. The FSM handles the addresses so that only bytes in the designated bounding rectangle are read or written.

This mode should probably be used even for read-modify-write access of large areas of the display since it will be much faster. In this case, the programmer can read a block of video memory into main RAM using INIR or INDR instructions, perform whatever modifications to the data that are required, and then write back to the hires RAM using OTIR or OTDR instructions.

The recommended procedure is as follows:

1. Write any value to the HR\_OP register with bit 7 reset ($00 will do). This step enables auto increment mode.
2. Write the ULX, ULY, LRX and LRY registers with values that will define the upper left and lower right corners of the desired rectangle. Writing to either or both the ULX or ULY registers will set the hires FSM address counter to an initial value based on those registers. It is recommended that the ULX and ULY registers be written in that order. Writing ULX and/or ULY initializes the hires memory address counter the FSM uses to access hires memory. This step MUST be performed or unexpected results may occur.
3. Write $A0 to the ULA address port at Z80 I/O address $BF3B.
4. Start reading OR writing to the data register by accessing the ULA data register at Z80 I/O port $FF3B. After a read or write, the FSM address counter will be incremented by one. This will have the effect of reading or writing the next pixel on the current line. If the FSM detects that the new column address is larger than the LRX value, the address will be automatically set to the first column of the next line. If the FSM detects that the next line would be below the LRY value, the memory pointer will be latched to the pixel at LRX, LRY.  
   NOTE: in this mode, ANY read or write will increment the hires RAM address so inadvertent accesses will advance the address counter to values that the programmer may not want.
5. Once the access has reached the final byte, further reads or writes will return the byte corresponding to the location at LRX, LRY.

### Z80 Access to Hires Buffer Memory

#### Non-auto Increment

The recommended procedure is as follows:

1. Write any value to the HR\_OP register with bit 7 reset ($00 will do). This step enables the auto increment mode.
2. Write the ULX, ULY with the desired pixel location
3. Write $AB to the ULA address port at Z80 I/O address $BF3B.
4. Read or write the hires byte as desired by accessing the ULA data port at Z80 I/O address $FF3B.
5. Repeat step 2 to 4 until all desired data is read/written.

#### Auto Increment

The recommended procedure is as follows:

1. Write any value to the HR\_OP register with bit 7 reset ($00 will do). This step enables auto increment mode.
2. Write the ULX, ULY, LRX and LRY registers with values that will define the upper left and lower right corners of the desired rectangle. Writing to either or both the ULX or ULY registers will set the hires FSM address counter to an initial value based on those registers. It is recommended that the ULX and ULY registers be written in that order. Writing ULX and/or ULY initializes the hires memory address counter the FSM uses to access hires memory. This step MUST be performed or unexpected results may occur.
3. Write $A0 to the ULA address port at Z80 I/O address $BF3B.
4. Start reading OR writing to the data register by accessing the ULA data register at Z80 I/O port $FF3B. After a read or write, the FSM address counter will be incremented by one. This will have the effect of reading or writing the next pixel on the current line. If the FSM detects that the new column address is larger than the LRX value, the address will be automatically set to the first column of the next line. If the FSM detects that the next line would be below the LRY value, the memory pointer will be latched to the pixel at LRX, LRY.  
   NOTE: in this mode, ANY read or write will increment the hires RAM address so inadvertent accesses will advance the address counter to values that the programmer may not want.
5. Once the access has reached the final byte, further reads or writes will return the byte corresponding to the location at LRX, LRY.

### Hires Video Registers

|  |  |  |
| --- | --- | --- |
| Reg # | Description | |
| $A0  HR\_DAT | Character port for text mode video display | |
| Mode: R/W | | Reset Value: $00 |
| Bits | | |
| 7..0 | Read or write to this register to set or get a memory value | |
| Note: If auto-increment is enabled, reading or writing to this register will the increment internal memory pointer to allow accessing the next byte of the defined rectangle. The memory pointer will not advance past the end of the rectangle | | |

|  |  |  |
| --- | --- | --- |
| Reg # | Description | |
| $A1  HR\_CMD | Command Register | |
| Mode: W | | Reset Value: $00 |
| Bits | | |
| Cmd # | Writing to this register will start a fill or copy command | |
| $00 | No operation | |
| $01 | Fill the defined rectangle with the value in register $A8 | |
| $02 | Copy the contents of the defined rectangle to the destination defined by registers $A6 and $A7 | |
|  |  | |
|  | | |
| $A1 | Command Register | |
| Mode: R | | Reset Value: $00 |
| Bits | | |
| 7..1 | zero | |
| 0 | ‘1’ – the FSM is busy with the current command | |

|  |  |  |
| --- | --- | --- |
| Reg # | Description | |
| $A2  ULX | Defines the upper left column pixel address of the defined rectangle | |
| Mode: R/W | | Reset Value: $00 |
| Bits | | |
| 7..0 | Value 0 to 255 | |
|  |  | |
| Notes:   1. Writing to this register will load the internal memory pointer 2. This register will accept any value between 0 and 255 3. The upper left corner of the display is $00 while the lower right corner is $FF | | |

|  |  |  |
| --- | --- | --- |
| Reg # | Description | |
| $A3  ULY | Defines the upper left row pixel address of the defined rectangle | |
| Mode: R/W | | Reset Value: $00 |
| Bits | | |
| 7..0 | Value 0 to 191 | |
|  |  | |
| Notes:   1. Writing to this register will load the internal memory pointer 2. This register will accept any value between 0 and 191 3. The upper left corner of the display is $00 while the lower right corner is $BF | | |

|  |  |  |
| --- | --- | --- |
| Reg # | Description | |
| $A4  LRX | Defines the lower right column pixel address of the defined rectangle | |
| Mode: R/W | | Reset Value: $00 |
| Bits | | |
| 7..0 | Value 0 to 255 | |
|  |  | |
| Notes:   1. This register will accept any value between 0 and 255 2. The upper left corner of the display is $00 while the lower right corner is $FF | | |

|  |  |  |
| --- | --- | --- |
| Reg # | Description | |
| $A5  LRY | Defines the lower right row pixel address of the defined rectangle | |
| Mode: R/W | | Reset Value: $00 |
| Bits | | |
| 7..0 | Value 0 to 191 | |
|  |  | |
| Notes:   1. This register will accept any value between 0 and 191 2. The upper left corner of the display is $00 while the lower right corner is $BF | | |

|  |  |  |
| --- | --- | --- |
| Reg # | Description | |
| $A6  DESTX | Defines the upper left column pixel address of the destination for a copy | |
| Mode: R/W | | Reset Value: $00 |
| Bits | | |
| 7..0 | Value 0 to 255 | |
|  |  | |
| Notes:   1. This register will accept any value between 0 and 255 2. The upper left corner of the display is $00 while the lower right corner is $FF | | |

|  |  |  |
| --- | --- | --- |
| Reg # | Description | |
| $A7  DESTY | Defines the upper left row pixel address of the destination for a copy | |
| Mode: R/W | | Reset Value: $00 |
| Bits | | |
| 7..0 | Value 0 to 191 | |
|  |  | |
| Notes:   1. This register will accept any value between 0 and 191 2. The upper left corner of the display is $00 while the lower right corner is $BF | | |

|  |  |  |
| --- | --- | --- |
| Reg # | Description | |
| $A8  HR\_FILL | Fill color | |
| Mode: R/W | | Reset Value: $00 |
| Bits | | |
| 7..0 | Color bits: ggrrbb | |
|  | | |

|  |  |  |
| --- | --- | --- |
| Reg # | Description | |
| $A9  HR\_OP | Operation value | |
| Mode: R/W | | Reset Value: $00 |
| Bits | | |
| 7..0 | Bit 7: if bit 7 is set, auto increment is disabled for reading/writing screen memory via HR\_DAT.  $00 – write the fill or source to the current destination memory address  $01 – bitwise AND the fill or source with the current value at the destination address. Write the result to the destination address  $02 – bitwise OR the fill or source with the current value at the destination address. Write the result to the destination address  $03 – bitwise XOR the fill or source with the current value at the destination address. Write the result to the destination address  $04 – write the bitwise inverted fill or source to the current memory address  $05 – bitwise NAND the fill or source with the current value at the destination address. Write the result to the destination address  $06 – bitwise NOR the fill or source with the current value at the destination address. Write the result to the destination address  $07 – bitwise XNOR the fill or source with the current value at the destination address. Write the result to the destination address | |
|  | | |
| Notes:   1. This allows simple bitwise operations to be performed during copies or fills | | |

|  |  |  |
| --- | --- | --- |
| Reg # | Description | |
| $AA  HR\_BORDER | Hires border color | |
| Mode: R/W | | Reset Value: $00 |
| Bits | | |
| 7..0 | Color bits: ggrrbb | |
|  | | |
| Notes:   1. This value is used for the screen border color whenever the hires screen is being used. | | |

|  |  |  |
| --- | --- | --- |
| Reg # | Description | |
| $AB  HR\_BDATA | Hires border color | |
| Mode: R/W | | Reset Value: $00 |
| Bits | | |
| 7..0 | Color bits: ggrrbb | |
|  | | |
| Note: If auto-increment is enabled, reading or writing to this register will the increment internal memory pointer to allow accessing the next byte of the defined rectangle. The memory pointer will not advance past the end of the rectangle | | |

# Accessing the ULA Plus Registers

Many of the video peripherals use the ULA Plus register space for control and configuration. Reading and writing these registers is relatively easy. Generally speaking, reading or writing a register is a two step process. The first step is to write the register number of the desired register to the ULA Plus address register. The programmer then writes or reads or writes the ULA Plus data register as required. This process is repeated for each register one wishes to access.

## Writing ULA Plus Registers

The following code shows one method to write the ULA Plus registers:

ld bc, $BF3B ; send the register address to the ULA Plus address register

ld a, $8E

out (c), a

ld b, $FF ; the I/O address is now $FF3B – the data register address

ld a, $80 ; value to write

out (c), a ; write the data to the ULA Plus register

## Reading ULA Plus Registers

The following code shows one method to write the ULA Plus registers:

ld bc, $BF3B ; send the register address to the ULA Plus address register

ld a, $73

out (c), a

ld b, $FF ; the I/O address is now $FF3B – the data register address

in a, (c) ; read the data from the ULA Plus register

## ULA Plus Registers

This is the full list of the ULA Plus registers. Any registers without a description are unused.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ULA Plus Palette | | | | | | | | | | | |
| Dec | Hex | Description | Dec | Hex | Description | Dec | Hex | Description | Dec | Hex | Description |
| 0 | 00 | Palette 0 | 32 | 20 | Palette 1 | 16 | 10 | Palette 2 | 48 | 30 | Palette 3 |
| 1 | 01 | 33 | 21 | 17 | 11 | 49 | 31 |
| 2 | 02 | 34 | 22 | 18 | 12 | 50 | 32 |
| 3 | 03 | 35 | 23 | 19 | 13 | 51 | 33 |
| 4 | 04 | 36 | 24 | 20 | 14 | 52 | 34 |
| 5 | 05 | 37 | 25 | 21 | 15 | 53 | 35 |
| 6 | 06 | 38 | 26 | 22 | 16 | 54 | 36 |
| 7 | 07 | 39 | 27 | 23 | 17 | 55 | 37 |
| 8 | 08 | 40 | 28 | 24 | 18 | 56 | 38 |
| 9 | 09 | 41 | 29 | 25 | 19 | 57 | 39 |
| 10 | 0A | 42 | 2A | 26 | 1A | 58 | 3A |
| 11 | 0B | 43 | 2B | 27 | 1B | 59 | 3B |
| 12 | 0C | 44 | 2C | 28 | 1C | 60 | 3C |
| 13 | 0D | 45 | 2D | 29 | 1D | 61 | 3D |
| 14 | 0E | 46 | 2E | 30 | 1E | 62 | 3E |
| 15 | 0F | 47 | 2F | 31 | 1F | 63 | 3F |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ULA Plus Control | | | | | | | | | | | |
| Dec | Hex | Description | Dec | Hex | Description | Dec | Hex | Description | Dec | Hex | Description |
| 64 | 40 | ULA\_SEL | 96 | 60 |  | 80 | 50 |  | 112 | 70 |  |
| 65 | 41 |  | 97 | 61 |  | 81 | 51 |  | 113 | 71 |  |
| 66 | 42 |  | 98 | 62 |  | 82 | 52 |  | 114 | 72 |  |
| 67 | 43 |  | 99 | 63 |  | 83 | 53 |  | 115 | 73 |  |
| 68 | 44 |  | 100 | 64 |  | 84 | 54 |  | 116 | 74 |  |
| 69 | 45 |  | 101 | 65 |  | 85 | 55 |  | 117 | 75 |  |
| 70 | 46 |  | 102 | 66 |  | 86 | 56 |  | 118 | 76 |  |
| 71 | 47 |  | 103 | 67 |  | 87 | 57 |  | 119 | 77 |  |
| 72 | 48 |  | 104 | 68 |  | 88 | 58 |  | 120 | 78 |  |
| 73 | 49 |  | 105 | 69 |  | 89 | 59 |  | 121 | 79 |  |
| 74 | 4A |  | 106 | 6A |  | 90 | 5A |  | 122 | 7A |  |
| 75 | 4B |  | 107 | 6B |  | 91 | 5B |  | 123 | 7B |  |
| 76 | 4C |  | 108 | 6C |  | 92 | 5C |  | 124 | 7C |  |
| 77 | 4D |  | 109 | 6D |  | 93 | 5D |  | 125 | 7D |  |
| 78 | 4E |  | 110 | 6E |  | 94 | 5E |  | 126 | 7E |  |
| 79 | 4F |  | 111 | 6F |  | 95 | 5F |  | 127 | 7F |  |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ULA and Hires Control | | | | | | | | | | | |
| Dec | Hex | Description | Dec | Hex | Description | Dec | Hex | Description | Dec | Hex | Description |
| 128 | 80 | MODE | 144 | 90 |  | 160 | A0 | HR\_DAT | 176 | B0 |  |
| 129 | 81 | COLOR | 145 | 91 |  | 161 | A1 | HR\_CMD | 177 | B1 |  |
| 130 | 82 | BLUL | 146 | 92 |  | 162 | A2 | HR\_ULX | 178 | B2 |  |
| 131 | 83 | IO\_BANK | 147 | 93 |  | 163 | A3 | HR\_ULY | 179 | B3 |  |
| 132 | 84 | MEMBANK | 148 | 94 |  | 164 | A4 | HR\_LRX | 180 | B4 |  |
| 133 | 85 | CHRADRL | 149 | 95 |  | 165 | A5 | HR\_LRY | 181 | B5 |  |
| 134 | 86 | CHRADRH | 150 | 96 |  | 166 | A6 | HR\_DESTX | 182 | B6 |  |
| 135 | 87 | CHRDATA | 151 | 97 |  | 167 | A7 | HR\_DESTY | 183 | B7 |  |
| 136 | 88 | EXPBORDER | 152 | 98 |  | 168 | A8 | HR\_FILL | 184 | B8 |  |
| 137 | 89 | EXPVIDMODE | 153 | 99 |  | 169 | A9 | HR\_OP | 185 | B9 |  |
| 138 | 8A | SPECCYBLACK | 154 | 9A |  | 170 | AA | HR\_BORDER | 186 | BA |  |
| 139 | 8B | COLOR16BANK | 155 | 9B |  | 171 | AB |  | 187 | BB |  |
| 140 | 8C | DFBANK | 156 | 9C |  | 172 | AC |  | 188 | BC |  |
| 141 | 8D | TRANSPCOLOR | 157 | 9D |  | 173 | AD |  | 189 | BD |  |
| 142 | 8E | VIDMODEA | 158 | 9E |  | 174 | AE |  | 190 | BE |  |
| 143 | 8F | VIDMODEB | 159 | 9F |  | 175 | AF |  | 191 | BF |  |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Unused Registers | | | | | | | | | | | |
| Dec | Hex | Description | Dec | Hex | Description | Dec | Hex | Description | Dec | Hex | Description |
| 192 | C0 |  | 224 | E0 |  | 208 | D0 |  | 240 | F0 |  |
| 193 | C1 |  | 225 | E1 |  | 209 | D1 |  | 241 | F1 |  |
| 194 | C2 |  | 226 | E2 |  | 210 | D2 |  | 242 | F2 |  |
| 195 | C3 |  | 227 | E3 |  | 211 | D3 |  | 243 | F3 |  |
| 196 | C4 |  | 228 | E4 |  | 212 | D4 |  | 244 | F4 |  |
| 197 | C5 |  | 229 | E5 |  | 213 | D5 |  | 245 | F5 |  |
| 198 | C6 |  | 230 | E6 |  | 214 | D6 |  | 246 | F6 |  |
| 199 | C7 |  | 231 | E7 |  | 215 | D7 |  | 247 | F7 |  |
| 200 | C8 |  | 232 | E8 |  | 216 | D8 |  | 248 | F8 |  |
| 201 | C9 |  | 233 | E9 |  | 217 | D9 |  | 249 | F9 |  |
| 202 | CA |  | 234 | EA |  | 218 | DA |  | 250 | FA |  |
| 203 | CB |  | 235 | EB |  | 219 | DB |  | 251 | FB |  |
| 204 | CC |  | 236 | EC |  | 220 | DC |  | 252 | FC |  |
| 205 | CD |  | 237 | ED |  | 221 | DD |  | 253 | FD |  |
| 206 | CE |  | 238 | EE |  | 222 | DE |  | 254 | FE |  |
| 207 | CF |  | 239 | EF |  | 223 | DF |  | 255 | FF |  |

# Example Code

## FPGA Definitions

// these registers are for general purpose use

#define FPGA\_TEST\_PORT 0xF0

#define FPGA\_REG 0xC3

// returns the FPGA version number

#define FPGA\_VER 0xD0

// Hardware stack ports

// read/write data from/to the stack

#define DOSSTK\_DAT 0xC0

// read/write

// read: get the current stack pointer value

// write: set the current stack pointer value

#define DOSSTKPTRL 0xC1

#define DOSSTKPTRH 0xC2

// SPI peripheral registers

// write: send a byte via the SPI peripheral

// read: get a byte via the SPI peripheral

#define SPI\_DREG 0xC8

// various SPI configuration bits

#define SPI\_SELREG 0xC9

#define BIT\_SPIMODE 7 // '1' sets SPI mode to 2 else SPI mode 0

#define MSK\_SPIMODE 1 << BIT\_SPIMODE

// masks to assert one of the four peripheral select lines

#define MSK\_PSEL0 0x40

#define MSK\_PSEL1 0x41

#define MSK\_PSEL2 0x42

#define MSK\_PSEL3 0x43

//

#define SPI\_STAT 0xCA

#define BIT\_SPI\_BUSY 0

#define BIT\_SPI\_MODE 7

#define NSK\_SPI\_BUSY 1 << BIT\_SPI\_BUSY

#define NSK\_SPI\_MODE 1 << BIT\_SPI\_MODE

// I/O address of the text mode peripheral

#define TEXT\_PORT 0xA2

// bit numbers for use by the Z80 bit instructions

#define BIT\_TXTSTAT\_DONE 1

#define BIT\_TXTSTAT\_BUSY 6

// masks for using AND/OR instructions

#define MSK\_TXTSTAT\_DONE 1 << BIT\_TXTSTAT\_DONE

#define MSK\_TXTSTAT\_BUSY 1 << BIT\_TXTSTAT\_BUSY

// I/O addresses to access the mamory mapper register set

#define MAP\_ADR 0xB0

// I/O address to read/write data to the mapper register set

#define MAP\_DAT 0x81

// MAP\_ADR value to access the mapper control register

#define REG\_MAPCTL 0x18

// mapper control register bits

// setting one or more of these bits enables the corresponding

// banks for mapping

// bit numbers for use by the Z80 bit instructions

#define BIT\_ENAHOME 0

#define BIT\_ENAEXROM 1

#define BIT\_ENADOCK 2

// masks for using AND/OR instructions

#define MSK\_ENAHOME (1 << BIT\_ENAHOME)

#define MSK\_ENAEXROM (1 << BIT\_ENAEXROM)

#define MSK\_ENADOCK (1 << BIT\_ENADOCK)

// MAP\_ADR values for the home mapping registers

#define HOME0 0x00

#define HOME1 0x01

#define HOME2 0x02

#define HOME3 0x03

#define HOME4 0x04

#define HOME5 0x05

#define HOME6 0x06

#define HOME7 0x07

// MAP\_ADR values for the home mapping registers

#define EXROM0 0x08

#define EXROM1 0x09

#define EXROM2 0x0A

#define EXROM3 0x0B

#define EXROM4 0x0C

#define EXROM5 0x0D

#define EXROM6 0x0E

#define EXROM7 0x0F

// dock mapping

#define DOCK0 0x10

#define DOCK1 0x11

#define DOCK2 0x12

#define DOCK3 0x13

#define DOCK4 0x14

#define DOCK5 0x15

#define DOCK6 0x16

#define DOCK7 0x17

// I/O addresses to access the ULA Plus register set

// This I/O space is used for the ULA Plus as well

// as other functions

#define ULA\_ADR 0xBF3B

#define ULA\_DAT 0xFF3B

#define ULA\_PALA0 0x00

#define ULA\_PALA1 0x01

#define ULA\_PALA2 0x02

#define ULA\_PALA3 0x03

#define ULA\_PALA4 0x04

#define ULA\_PALA5 0x05

#define ULA\_PALA6 0x06

#define ULA\_PALA7 0x07

#define ULA\_PALA8 0x08

#define ULA\_PALA9 0x09

#define ULA\_PALAA 0x0A

#define ULA\_PALAB 0x0B

#define ULA\_PALAC 0x0C

#define ULA\_PALAD 0x0D

#define ULA\_PALAE 0x0E

#define ULA\_PALAF 0x0F

#define ULA\_PALB0 0x10

#define ULA\_PALB1 0x11

#define ULA\_PALB2 0x12

#define ULA\_PALB3 0x13

#define ULA\_PALB4 0x14

#define ULA\_PALB5 0x15

#define ULA\_PALB6 0x16

#define ULA\_PALB7 0x17

#define ULA\_PALB8 0x18

#define ULA\_PALB9 0x19

#define ULA\_PALBA 0x1A

#define ULA\_PALBB 0x1B

#define ULA\_PALBC 0x1C

#define ULA\_PALBD 0x1D

#define ULA\_PALBE 0x1E

#define ULA\_PALBF 0x1F

#define ULA\_PALC0 0x20

#define ULA\_PALC1 0x21

#define ULA\_PALC2 0x22

#define ULA\_PALC3 0x23

#define ULA\_PALC4 0x24

#define ULA\_PALC5 0x25

#define ULA\_PALC6 0x26

#define ULA\_PALC7 0x27

#define ULA\_PALC8 0x28

#define ULA\_PALC9 0x29

#define ULA\_PALCA 0x2A

#define ULA\_PALCB 0x2B

#define ULA\_PALCC 0x2C

#define ULA\_PALCD 0x2D

#define ULA\_PALCE 0x2E

#define ULA\_PALCF 0x2F

#define ULA\_PALD0 0x30

#define ULA\_PALD1 0x31

#define ULA\_PALD2 0x32

#define ULA\_PALD3 0x33

#define ULA\_PALD4 0x34

#define ULA\_PALD5 0x35

#define ULA\_PALD6 0x36

#define ULA\_PALD7 0x37

#define ULA\_PALD8 0x38

#define ULA\_PALD9 0x39

#define ULA\_PALDA 0x3A

#define ULA\_PALDB 0x3B

#define ULA\_PALDC 0x3C

#define ULA\_PALDD 0x3D

#define ULA\_PALDE 0x3E

#define ULA\_PALDF 0x3F

#define ULA\_ENA 0x40

#define BIT\_ULA\_ENA 0

#define MSK\_ULA\_ENA (1 << ULA\_ENA)

#define UR\_EXMODE 0x80

// set this bit to enable the extended modes

#define BIT\_ENAEXT 7

// extended mode standard 64 column attribute

#define MSK\_ENAEXT (1 << BIT\_ENAEXT)

#define MSK\_XINKBLACK (0x0 << 3)

#define MSK\_XINKBLUE (0x1 << 3)

#define MSK\_XINKRED (0x2 << 3)

#define MSK\_XINKMAGEN (0x3 << 3)

#define MSK\_XINKGREEN (0x4 << 3)

#define MSK\_XINKCYAN (0x5 << 3)

#define MSK\_XINKYELL (0x6 << 3)

#define MSK\_XINKWHITE (0x7 << 3)

// extended mode vide modes

#define MSK\_XMODES0 0x00 // display file 1

#define MSK\_XMODES1 0x00 // display file 2

#define MSK\_XHICLR 0x02 // high color mode

#define MSK\_XHIRES 0x06 // 512x192 hires mode

#define MSK\_XHIRESP 0x0E // 512x192 hires with attributes

#define MSK\_XCOLOR4 0x0F // 4 color 256x192 mode

#define MSK\_XCOLOR16 0x0B // 16 color 128 x 192 mode

#define UR\_COLOR 0x81 // upper 8 bits of the 9 bit ULA color value

#define UR\_BLULOW 0x82 // low bit of the ULA blue

#define UR\_EXPBORD 0x83 // expanded mode border color

#define UR\_SPECYBLK 0x84 // '1' enables Spectrum black (only one shade of black)

#define UR\_CLR4BNK 0x85 // upper 4 bits to access the ULA bank for 4 bit color

// transparent color byte: gggrrrbb

#define UR\_TRANSCLR 0x86

// selects the video output mux control

#define UR\_VMODEA 0x87

#define UR\_VMODEB 0x88

#define MODETS 0x00 // select 2068 video

#define MODEHR 0x01 // select hires videl

#define MODETH 0x02 // timex foreground, hires background

#define MODEHT 0x03 // hires foreground, timex background

#define MODEVDP 0x40 // select VDP9918

#define MODETXT 0x80 // select text mode

#define UR\_HRDAT 0xA0 // port to read/write to hires screen buffer

#define UR\_HRCMD 0xA1 // hires command. writing to this port executes the command

#define HRC\_NOP 0x00

#define HRC\_FILL 0x01

#define HRC\_COPY 0x02

#define HRC\_BTOS 0x03

#define HRC\_STOB 0x04

// x and y coordinates of the upper left

// corner of the desired rectangle

#define UR\_HRULX 0xA2

#define UR\_HRULY 0xA3

// x and y coordinates of the lower right

// corner of the desired rectangle

#define UR\_HRLRX 0xA4

#define UR\_HRLRY 0xA5

// x and y coordinates of the upper left

// corner of the destination of a copy

// command

#define UR\_HRDESTX 0xA6

#define UR\_HRDESTY 0xA7

// fill color

#define UR\_HRFILL 0xA8

// set operation for fill and copy

// commands

#define UR\_HROP 0xA9

#define HRNOP 0x00

#define HRAND 0x01

#define HROR 0x02

#define HRXOR 0x03

#define HRINV 0x04

#define HRNAND 0x05

#define HRNOR 0x06

#define HRXNOR 0x07

// inhibits auto increment when set

#define HRC\_INHINC 0x80

#define UR\_HRBORDER 0xAA // border color for hires

#define UR\_HRBDAT 0xAB // read/write buffer memory.

## COLOR4 Example

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

#include "fpga.h"

void writeUlaPaletteA(void);

void writeUlaReg(unsigned char reg, val);

void clearScreen(unsigned int clrColor) \_\_z88dk\_fastcall;

void setColor4Mode(void);

void setColor16Mode(void);

void writeTestBlks(void);

void setPixel(unsigned int x, y, unsigned char color);

void writeOneTestBlock(unsigned int x, y);

// ULA Plus palette color definitions

unsigned char ulaPA[16] = { 0x00, 0x03, 0x1c, 0x1f,

0xe0, 0xe3, 0xfc, 0xff,

0x49, 0x01, 0x0c, 0x0e,

0x60, 0x61, 0x76, 0x6d};

// definition for a 2X8 test block in packed format

unsigned char testCB[8] = { 0x1B, 0x00, 0x55, 0xAA,

0xFF, 0x05, 0x5F, 0xB1};

int main()

{

// write the palett values to the ULA Plus registers

writeUlaPaletteA();

// set the border color for our mode

writeUlaReg(UR\_EXPBORD, 0xfc);

// set the 16 color mode

setColor4Mode();

// clear the screen

clearScreen(0x03);

writeTestBlks();

// loop here since the video mode is incompatible

// with the Timex display and we won't be able to

// see anything if dropped back to the command line

while(1);

}

void writeUlaPaletteA(void){

unsigned char i;

for (i = ULA\_PALA0; i < ULA\_PALB0; i++){

writeUlaReg(i, ulaPA[i]);

}

}

void clearScreen(unsigned int clrColor) \_\_z88dk\_fastcall

{

#asm

; this section makes the color value a proper

; 8 bit value to write to memory to clear the screen

ld a, l ; get the color value

and $03 ; keep only the 4 lower bits, clear carry

ld b, a ; save the color value

rla ; shift right 4 bits

rla

or b

ld b, a

rla

rla

or b ; the color value is now ready in A

ld b, a

rla

rla

or b ; the color value is now ready in A

; we clear the screen by writing the color value to

; each byte of DF0...

ld hl, $4000 ; DF0 address

ld de, $4001

ld bc, 6143

ld (hl), a

ldir

; ... and repeating for DF1

ld hl, $6000

ld de, $6001

ld bc, 6143

ld (hl), a

ldir

#endasm

}

void writeTestBlks(void){

unsigned int i, j;

unsigned char \*adr;

// write a test block to the upper left corner of DF0

writeOneTestBlock(0, 0);

// write a test block to the upper right corner of DF0

writeOneTestBlock(252, 0);

// write a test block to the lower left corner of DF0

writeOneTestBlock(0, 88);

// write a test block to the lower right corner of DF0

writeOneTestBlock(252, 88);

// write a test block to the upper left corner of DF1

writeOneTestBlock(0, 96);

// write a test block to the upper right corner of DF1

writeOneTestBlock(252, 96);

// write a test block to the lower left corner of DF1

writeOneTestBlock(0, 184);

// write a test block to the lower right corner of DF1

writeOneTestBlock(252, 184);

// write a test block that spans DF1 and DF2.

// we write near the left side so we can use the

// existing blocks as rulers

writeOneTestBlock(6, 94);

}

//-----------------------------------------------------

// Write a test block at the pixel coordinates x, y

// x and y specify the upper left corner of the

// test block

//

void writeOneTestBlock(unsigned int x, y){

unsigned char i;

unsigned char pixel;

// loop for the 8 vertical color bars

for (i = 0; i < 8; i++){

// get the even pixel color...

pixel = testCB[i] >> 6;

// ... and write it

setPixel(x, y, pixel);

// advance to the next column

x += 1;

// get the even pixel color...

pixel = testCB[i] >> 4;

// ... and write it

setPixel(x, y, pixel);

// advance to the next column

x += 1;

// get the even pixel color...

pixel = testCB[i] >> 2;

// ... and write it

setPixel(x, y, pixel);

// advance to the next column

x += 1;

// get the odd pixel color...

pixel = testCB[i] & 0x03;

// ... and set it

setPixel(x, y, pixel);

// back to the first column

x -= 3;

// next line

y += 1;

}

}

//-----------------------------------------------------

// Set a pixel at x, y with "color"

// this routine silently returns if the pixel location

// is invalid

//

void setPixel(unsigned int x, y, unsigned char color){

unsigned char \*adr;

unsigned int baseAdr;

unsigned char pixel;

// ensure our pixel stays within the

// screen bounds

if (y > 191) return;

if (x > 255) return ;

// clip the color value

color &= 0x03;

// if the pixel is even, shift it left by 4 bits

switch (x & 3){

case 0:

color = color << 6;

break;

case 1:

color = color << 4;

break;

case 2:

color = color << 2;

break;

}

// set the base address of the block based on

// the y coordinate

baseAdr = 16384;

if (y > 95){

// in DF1

baseAdr += 8192;

// make compatible with the succeeding computations

y -= 96;

}

// compute the address of the pixel

// there are 64 bytes per scan line, the left shift

// by 6 does this

adr = (unsigned char \*)((y << 6) + (x >> 2) + baseAdr);

// get the current

pixel = \*adr;

//

switch (x & 3){

case 0:

pixel &= 0x3F;

break;

case 1:

pixel &= 0xCF;

break;

case 2:

pixel &= 0xF3;

break;

case 3:

pixel &= 0xFC;

break;

}

// put the new pixel color in place

pixel |= color;

// and write to video memory

\*adr = pixel;

}

void setColor4Mode(void){

// set video A to TS vid

writeUlaReg(UR\_VMODEA, MODETS);

// enable advanced video and set to color4

writeUlaReg(UR\_EXMODE, (MSK\_ENAEXT | MSK\_XINKBLUE | MSK\_XCOLOR4));

}

// write a value to a ULA register

void writeUlaReg(unsigned char reg, val){

outp(ULA\_ADR, reg);

outp(ULA\_DAT, val);

}

## COLOR16 Example

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

#include "fpga.h"

void writeUlaPaletteA(void);

void writeUlaReg(unsigned char reg, val);

void clearScreen(unsigned int clrColor) \_\_z88dk\_fastcall;

void setColor4Mode(void);

void setColor16Mode(void);

void writeTestBlks(void);

void setPixel(unsigned int x, y, unsigned char color);

void writeOneTestBlock(unsigned int x, y);

// ULA Plus palette color definitions

unsigned char ulaPA[16] = { 0x00, 0x03, 0x1c, 0x1f,

0xe0, 0xe3, 0xfc, 0xff,

0x49, 0x01, 0x0c, 0x0e,

0x60, 0x61, 0x76, 0x6d};

// definition for a 2X8 test block in packed format

unsigned char testCB[8] = { 0x01, 0x23, 0x45, 0x67,

0x89, 0xab, 0xcd, 0xef};

int main()

{

// write the palett values to the ULA Plus registers

writeUlaPaletteA();

// set the border color for our mode

writeUlaReg(UR\_EXPBORD, 0xfc);

// set the 16 color mode

setColor16Mode();

// clear the screen

clearScreen(0x03);

writeTestBlks();

// loop here since the video mode is incompatible

// with the Timex display and we won't be able to

// see anything if dropped back to the command line

while(1);

}

// fill ULA palette #0 with 16 color values

void writeUlaPaletteA(void){

unsigned char i;

for (i = ULA\_PALA0; i < ULA\_PALB0; i++){

writeUlaReg(i, ulaPA[i]);

}

}

void clearScreen(unsigned int clrColor) \_\_z88dk\_fastcall

{

#asm

; this section makes the color value a proper

; 8 bit value to write to memory to clear the screen

ld a, l ; get the color value

and $0F ; keep only the 4 lower bits, clear carry

ld b, a ; save the color value

rla ; shift right 4 bits

rla

rla

rla

or b ; the color value is now ready in A

; we clear the screen by writing the color value to

; each byte of DF0...

ld hl, $4000 ; DF0 address

ld de, $4001

ld bc, 6143

ld (hl), a

ldir

; ... and repeating for DF1

ld hl, $6000

ld de, $6001

ld bc, 6143

ld (hl), a

ldir

#endasm

}

void writeTestBlks(void){

unsigned int i, j;

unsigned char \*adr;

// write a test block to the upper left corner of DF0

writeOneTestBlock(0, 0);

// write a test block to the upper right corner of DF0

writeOneTestBlock(126, 0);

// write a test block to the lower left corner of DF0

writeOneTestBlock(0, 88);

// write a test block to the lower right corner of DF0

writeOneTestBlock(126, 88);

// write a test block to the upper left corner of DF1

writeOneTestBlock(0, 96);

// write a test block to the upper right corner of DF1

writeOneTestBlock(126, 96);

// write a test block to the lower left corner of DF1

writeOneTestBlock(0, 184);

// write a test block to the lower right corner of DF1

writeOneTestBlock(126, 184);

// write a test block that spans DF1 and DF2.

// we write near the left side so we can use the

// existing blocks as rulers

writeOneTestBlock(3, 94);

}

//-----------------------------------------------------

// Write a test block at the pixel coordinates x, y

// x and y specify the upper left corner of the

// test block

//

void writeOneTestBlock(unsigned int x, y){

unsigned char i;

unsigned char pixel;

// loop for the 8 vertical color bars

for (i = 0; i < 8; i++){

// get the even pixel color...

pixel = testCB[i] >> 4;

// ... and write it

setPixel(x, y, pixel);

// advance to the next column

x += 1;

// get the odd pixel color...

pixel = testCB[i] & 0x0F;

// ... and set it

setPixel(x, y, pixel);

// back to the first column

x -= 1;

// next line

y += 1;

}

}

//-----------------------------------------------------

// Set a pixel at x, y with "color"

// this routine silently returns if the pixel location

// is invalid

//

void setPixel(unsigned int x, y, unsigned char color){

unsigned char \*adr;

unsigned int baseAdr;

unsigned char pixel;

// ensure our pixel stays within the

// screen bounds

if (y > 191) return;

if (x > 127) return ;

// clip the color value

color &= 0x0F;

// if the pixel is even, shift it left by 4 bits

if ((x & 1) == 0) color = color << 4;

// set the base address of the block based on

// the y coordinate

baseAdr = 16384;

if (y > 95){

// in DF1

baseAdr += 8192;

// make compatible with the succeeding computations

y -= 96;

}

// compute the address of the pixel

// there are 64 bytes per scan line, the left shift

// by 6 does this

adr = (unsigned char \*)((y << 6) + (x >> 1) + baseAdr);

// get the current

pixel = \*adr;

//

if (x & 1){

// we are setting the odd pixel, so clear those bits

pixel &= 0xF0;

} else {

// we are setting the even pixel, so clear those bits

pixel &= 0x0F;

}

// put the new pixel color in place

pixel |= color;

// and write to video memory

\*adr = pixel;

}

void setColor4Mode(void){

// set video A to TS vid

writeUlaReg(UR\_VMODEA, MODETS);

// enable advanced video and set to color4

writeUlaReg(UR\_EXMODE, (MSK\_ENAEXT | MSK\_XINKBLUE | MSK\_XCOLOR4));

}

void setColor16Mode(void){

// set video A to TS vid

writeUlaReg(UR\_VMODEA, MODETS);

// enable advanced video and set to color16

writeUlaReg(UR\_EXMODE, (MSK\_ENAEXT | MSK\_XINKBLUE | MSK\_XCOLOR16));

}

// write a value to a ULA register

void writeUlaReg(unsigned char reg, val){

outp(ULA\_ADR, reg);

outp(ULA\_DAT, val);

}

The memory mapper provides access by the TS2068 to the 512k byes of external memory. The mapper uses 8K chunks just as the native TS2068 does but maps them in in a different manner.