

Z80 Project Mark 2: Documentation

Nathan Dumont

December 7, 2010

Contents

0.1	License	6
0.2	Revisions	6
1	Z80 I/O Ports	7
1.1	RCSTA - 0x01R	7
1.2	PIF - 0x06R	7
1.3	PIE - 0x06W	8
1.4	STATUS - 0x08R	9
2	Hardware Layout	11
2.1	UART	11
2.1.1	Jumpers	11
3	GPU Commands	15
3.1	GPU Ports	15
3.1.1	GPU_DATA	15
3.1.2	GPU_COMMAND	15
3.2	GPU Modes	15
4	Keyboard Controller	19
4.1	Keyboard Controller Commands	19
4.2	Translation Modes	19
4.3	ASCII Mode Codes	19
4.3.1	Control Keys	21
4.3.2	Media Keys	22
4.3.3	ACPI Control Keys	22
5	PIC Pin Allocations	23
6	PIC Source Structure	25
6.1	Files	25
6.1.1	Main (main.asm)	25
6.1.2	Serial (serial.asm)	25
6.1.3	Z80 Bus (host_bus.asm)	26
6.1.4	SD Card Functions (sd_card.asm)	27
6.1.5	Z80 Boot (boot.asm)	28
6.1.6	Boot ROM (rom.asm)	28

7	Debug Comms Protocol	29
7.1	Buffer Locations	29
7.2	Packet Specifications	29
7.2.1	Host to Device Packet Definition	29
7.2.2	Device to Host Packet Definition	30
7.3	Response Codes	30
7.4	Command Codes	30
7.4.1	Summary	31
7.4.2	Mem Block Write	32
7.4.3	BIOS Update	32
7.4.4	Do Command	32
7.5	Error Messages	32
7.6	seriallib.py	33
7.6.1	Packet	33
8	PIC BIOS Commands	35
8.1	Protocol	35
8.2	Commands	35
8.2.1	BIOS_RESET_CMD \$3F	35
8.2.2	BIOS_READ_VAR_CMD \$0A	35
8.2.3	BIOS_WRITE_VAR_CMD \$0B	36
8.2.4	SD_CARD_CID \$20	36
8.2.5	SD_CARD_CSD \$21	36
8.2.6	SD_CARD_READ_BLOCK \$22	36
9	Z80 BIOS Routines	37
9.1	Alphabetical List of Functions	38
9.1.1	bios_disable_int	38
9.1.2	bios_enable_int	39
9.1.3	bios_error_exit	39
9.1.4	bios_load_var	39
9.1.5	bios_reset	39
9.1.6	bios_save_var	39
9.1.7	bios_set_interrupts	40
9.1.8	fat_init	40
9.1.9	fat_next_block	40
9.1.10	fat_next_cluster	40
9.1.11	fat_select_cluster	40
9.1.12	gpu_cls	41
9.1.13	gpu_dec	41
9.1.14	gpu_get_colour	41
9.1.15	gpu_hex	41
9.1.16	gpu_init	42
9.1.17	gpu_set_colour	42
9.1.18	gpu_str	42
9.1.19	maths_add32	42
9.1.20	maths_asl32	43
9.1.21	maths_asln32	43
9.1.22	maths_asr32	43
9.1.23	maths_asrn32	44
9.1.24	maths_bcd_to_bin	44

9.1.25	maths_bin_to_bcd	44
9.1.26	maths_div32	44
9.1.27	maths_mod	44
9.1.28	maths_sub32	45
9.1.29	maths_test_z32	45
9.1.30	maths_test_z64	45
9.1.31	uart_write	45
9.1.32	usb_get_byte	45
9.1.33	usb_write_byte	46
9.2	Alphabetical List of Variables	46
9.2.1	gpu_hex_lut	46
9.2.2	maths_flags	47
9.2.3	maths_op_a	47
9.2.4	maths_op_b	47
9.2.5	maths_op_c	47
9.2.6	ram_top	47
9.2.7	stack	47
9.2.8	system_boot_device	48
9.2.9	system_boot_file	48
9.2.10	system_boot_order	48
9.2.11	system_clock	48
9.2.12	system_clock_speeds	49
9.2.13	system_filesystem	49
9.3	Functions and Variables by Source File	49
9.3.1	kb_commands.z8a	49
9.3.2	usb.z8a	50
9.3.3	maths.z8a	50
9.3.4	sd_commands.z8a	50
9.3.5	statics.z8a	51
9.3.6	gpu.z8a	51
9.3.7	uart.z8a	51
9.3.8	fat.z8a	51
9.3.9	sd.z8a	52
9.3.10	rtc.z8a	52
9.3.11	menu.z8a	52
9.3.12	gpu_commands.z8a	52
9.3.13	boot.z8a	52
9.3.14	bios.z8a	53
9.3.15	scancodes.z8a	53
9.4	Functions by Memory Location	54
9.5	Variables by Memory Location	55
10	File Formats	57
10.1	Assembly files *.z8a	57
10.2	Bootable files *.z8b	57
10.3	Program files *.z8p	57
10.4	Library files *.z8l	58
10.5	Executable Files	58

0.1 License

Z80 Project Mark 2: Documentation

Copyright (C) 2009 Nathan Dumont

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>

0.2 Revisions

- 13-Oct-2009 First creation of this manual. Previous documentation on www.nathandumont.com.
- 17-Oct-2009 Documentation of external functions and variables added to the sections on main.asm, serial.asm and host_bus.asm.
- 11-Dec-2009 Updated the serial commands and source documentation, also added info on the python library.
- 09-Jan-2010 Changed documentation to cover whole project. Started adding port and pin details for the Z80 bus peripherals.
- 05-Feb-2010 Started adding updates about the new hardware on the UI board.

Chapter 1

Z80 I/O Ports

The Z80's IO bus is fully decoded so all 256 ports are potentially available for read and write. Only a small number have been assigned in this version of the project however. These are summarised in Table 1.1. All the port definitions, and bit names where applicable are to be found in the `statics.z8a` source file which specifies any constants used in the BIOS code.

1.1 RCSTA - 0x01R

RCSTA is the UART status register. The meaning of the bits is explained below, mainly it specifies the status of the receiver, but there are two flags to throttle loading of the UART so that new bytes are not loaded before the old ones have been sent. For more details see the data sheet for a 6402 UART.

N/A	N/A	TRE	TBRE	PE	FE	OE	DR
7 msb	6	5	4	3	2	1	0 lsb

Bits	Name	Description
7-6	N/A	Unused
5	TRE	Set high once the whole character has been sent (including stop bits).
4	TBRE	Transmit buffer register empty when set high.
3	PE	A parity error occurred in the last reception when high.
2	FE	A frame error occurred (stop bit missing) when high.
1	OE	Over-run error, RCREG was not read before the new byte arrived when high.
0	DR	There is new data ready to read when high. This is used as an interrupt source for the Z80.

1.2 PIF - 0x06R

PIF contains a set of eight flag bits indicating the status of the eight configurable interrupt sources in the system. If a line is low that means that device is currently requesting an interrupt service. All eight bits of this port are combined with a bitwise and to produce the Z80's interrupt signal. A bit may be low but not caused an interrupt if the interrupt enable flip-flop on the Z80 is not

set. More than one bit may be low at once, the priority is from bit 0 to bit 7 so if both interrupt 2 and 3 are active at the same time 2 should be serviced first. Particular interrupts can be masked using the PIE register, if they are masked they will show as inactive in PIF, regardless of the state of the line coming from the device.

IF7	IF6	IF5	IF4	GPUIF	UARTIF	KBIF	CKIF
7 msb	6	5	4	3	2	1	0 lsb

Bits	Name	Description
7	IF7	Interrupt 7 (currently unassigned)
6	IF6	Interrupt 6 (currently unassigned)
5	IF5	Interrupt 5 (currently unassigned)
4	IF4	Interrupt 4 (currently unassigned)
3	GPUIF	GPU Interrupt, indicates when the GPU has booted
2	UARTIF	UART Interrupt, triggered when the UART receives a byte.
1	KBIF	Keyboard interrupt, a key press is ready or a connect event etc.
0	CKIF	Clock interrupt, a timer based interrupt from the RTC has occurred.

1.3 PIE - 0x06W

The PIE register is used to enable/disable specific device interrupts. The bits of the PIE register are bitwise ORed with the interrupt lines from their respective devices. This means that to enable an interrupt from a device (which is active low) you need to set the corresponding bit low in the PIE register.

IE7	IE6	IE5	IE4	GPUIE	UARTIE	KBIE	CKIE
7 msb	6	5	4	3	2	1	0 lsb

Bits	Name	Description
7	IE7	0 = allow interrupt 7 1 = disable interrupt 7
6	IE6	0 = allow interrupt 6 1 = disable interrupt 6
5	IE5	0 = allow interrupt 5 1 = disable interrupt 5
4	IE4	0 = allow interrupt 4 1 = disable interrupt 4
3	GPUIE	0 = allow GPU interrupt 1 = disable GPU interrupts
2	UARTIE	0 = allow UART interrupts 1 = disable UART interrupt
1	KBIE	0 = allow keyboard interrupts 1 = disable keyboard interrupt
0	CKIE	0 = allow RTC interrupts 1 = disable RTC interrupts



In the header file `statics.z8a` the IF bits are defined as the bit index e.g. CKIF is declared as 0 and GPUIF is declared as 3. This is suitable for use with the Z80 `bit`, `set` and `res` commands. The IE bits are declared as 8 bit masks, e.g. CKIE is 0x01, GPUIE 0x04. This makes them suitable for masks e.g. the start up value of the register can be done using the flags bitwise ORed by the compiler e.g. `UARTIE | KBIE` will mask off both the UART and Keyboard interrupts.

1.4 STATUS - 0x08R

The status register is used to return single bit status information from a number of the peripherals on the UI board. The bits are read only and are controlled by their related peripheral devices.

USB_RXF	USB_TXE	GPU_INT	KEY_DETECT	KEY_RDY	N/A	N/A	N/A
7 msb	6	5	4	3	2	1	0 lsb

Bits	Name	Description
7	USB_RXF	1 means there is data in the USB FIFO to read. 0 means there is no data, do not read from the USB FIFO.
6	USB_TXE	1 means it is safe to write to the USB FIFO. 0 means the write FIFO is full, don't write.
5	GPU_INT	This is an un-masked copy of the GPU interrupt line, so even if it is masked in the PIE register you can read the value here.
4	KEY_DETECT	This line is set (high, 1) if a keyboard is detected and provides a successful start up message (0xAA).
3	KEY_RDY	Indicates the state of the keyboard controller. This is set (high, 1) once the keyboard controller is running and ready.
2-0	N/A	Not used.

Address	Access	Name	Description
0x00	r	RCREG	UART receive port.
0x00	w	TXREG	UART transmit port.
0x01	r	RCSTA	UART Status port.
0x01	w	DEBUG	8 bit latch which drives LEDs for debug purposes.
0x02	r/w	SD_DATA	Access to the SD via the PIC while it is in slave mode.
0x03	r/w	KEY_DATA	Keyboard port, FIFO on read, accepts single commands on write.
0x04	r/w	GPU_COMMAND	Send commands to the GPU by writing if a response is available for that command read from this port.
0x05	r/w	GPU_DATA	Send characters to the screen, reads back character at cursor.
0x06	r	PIF	Peripheral interrupt flags, read to find the source of an interrupt.
0x06	w	PIE	Peripheral interrupt enable, used to mask some peripheral interrupt signals.
0x07	r/w	USB	Access to the Vinculum USB host chip.
0x08	r	STATUS	Access to various flags including USB FIFO status etc.

Table 1.1: Z80 I/O port functions

Chapter 2

Hardware Layout

2.1 UART

2.1.1 Jumpers

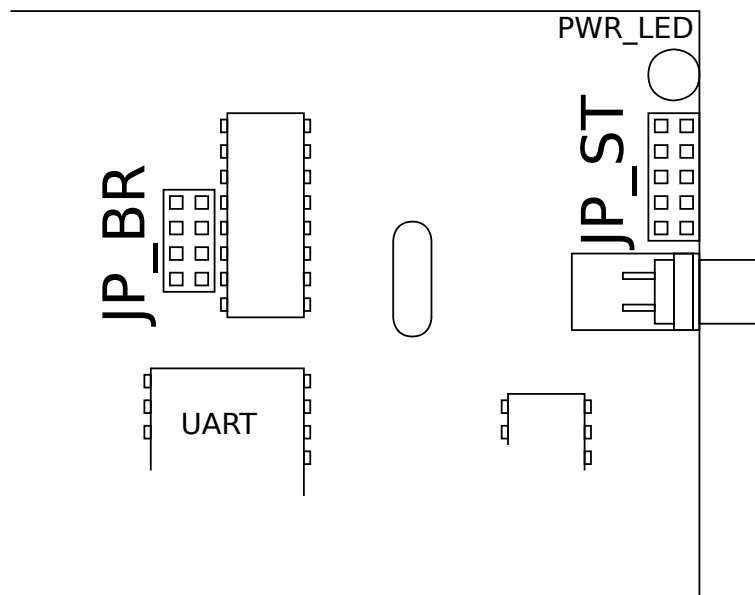


Figure 2.1: Jumper locations for UART settings

There are two sets of jumper headers that affect the UART, these are JP_BR and JP_ST shown in Figure 2.1. These headers are on the lower board near the serial port and the UART chip itself. The settings are detailed below.

JP_BR: Baud Rate

JP_BR connects the clock source to the UART. This allows the baud-rate of the communications to be selected from 4 possible speeds. Table 2.1 shows the possible speed settings.

IMPORTANT NOTE: It is essential to fit only one jumper in one of the four positions shown on JP_BR, connecting more will short out the clock chip's outputs!



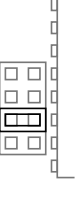
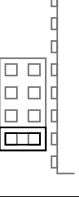
JP_BR 	4800
JP_BR 	9600
JP_BR 	2400
JP_BR 	19200

Table 2.1: Baud rate jumper positions

JP_ST: UART settings

The five jumpers on JP_ST can be fitted in a variety of combinations to affect the character length, parity settings and stop bits. Table 2.2 describes what the individual settings do. Some of the jumpers work in combination. In the table, positions which are shaded have no affect on the current setting, positions which are not shaded must be either fitted or not fitted as shown.






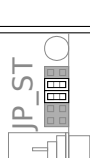
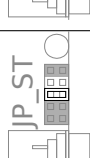
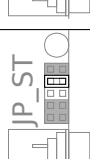

	No parity.
	Even parity.
	Odd parity.
	One stop bit.
	1.5 stop bits for character length of 5, 2 stop bits for all other lengths.
	5 bit character.
	6 bit character.
	7 bit character.
	8 bit character.

Table 2.2: UART configuration jumpers.

Chapter 3

GPU Commands

3.1 GPU Ports

The GPU has two ports on the IO bus, both are read/writeable. These are referred to as GPU_COMMAND (0x04) and GPU_DATA (0x05). The function of the GPU_DATA port has a constantly defined function, the command port accepts a set of pre-defined commands which affect the way that the GPU functions.

3.1.1 GPU_DATA

The GPU_DATA port accepts data bytes and writes them to the active buffer. By default this is a display buffer so the bytes should be ASCII character codes corresponding to characters to be displayed on the screen. Other data modes handle data in different ways, see the descriptions of these modes associated with their commands. By default all writes are done in “overwrite” mode, i.e. whatever data is currently at the cursor is replaced with the byte written, it is not shifted on, the cursor is automatically incremented.

Reading the data port will return a character from the cursor within the active buffer.

3.1.2 GPU_COMMAND

The GPU_COMMAND port takes byte-wide commands which are used to set variables within the GPU, and switch modes, e.g. select colour, row, or active screen. GPU commands are basically 4 bit long in the upper nibble of the byte. The lower four bits are used to convey bits of data to the GPU or flags depending on what the command is. A summary of commands is in Table 3.1 and details follow.

3.2 GPU Modes

The GPU can display output in different resolutions depending on the hardware attached. These are called modes. Two commands affect the mode setting and get info about the current and available modes. The GPU_MODE command provides the ability to set or get the current mode on the VGA output. The GPU_AVAILABLE_MODES command returns the highest mode that the current monitor has been detected to support. To allow for incorrect monitor detection, the GPU does not prevent selection of a mode that has not been detected as supported, it is advised that the software provides a prompt if deliberately exceeding the recommended highest setting and reverts if no input was received in a time period.

Bits	Name	Description
0000_0000	GPU_RESET	Issues a reset, clears all screens sets colour to 0 on all screens and sets both TV and VGA to virtual screen 1.
0001_0000	GPU_CLS	Clear screen, clears the current screen and sets cursor to (0,0). The colour is not affected.
0010_bccc	GPU_COLOUR	If b is set this sets the colour on the current screen to colour ccc (colours are 0-7).
0011_b000 n	GPU_COLUMN	If b is set this sets the column on the current screen to the next byte sent to the command port (n).
0100_b000 n	GPU_ROW	If b is set this sets the row on the current screen to the value of the next byte sent to the command port (n).
0101_0yyy	GPU_SET_VGA	Set the VGA to view virtual screen yyy. (Only screens 0 to 3 are implemented so far.)
0101_1yyy	GPU_SET_TV	Set the TV to view virtual screen yyy. (Only screens 0 to 3 are implemented so far.)
0110_yyyy	GPU_SELECT_SCREEN	Set the active screen to virtual screen number yyyy. (Currently only screens 0 to 3 are implemented, screens above 7 are intended to be non-display buffers)
0111_bmmm	GPU_MODE	Set the GPU mode (VGA output only, see Section 3.2 for details). If b is set then the mode is set to mmm, if b is 0 then the mode is available for read from the command port.
1000_0000	GPU_AVAILABLE_MODES	Get the highest mode currently detected to be supported by the attached screen.

Table 3.1: GPU Command summary

Mode	Resolution (Chars)	Description
0	512x480 (32x15)	Low res-mode, matches TV output. 16x16 or 16x32 pixel tiles, 4 colours per tile. Suitable for tile based graphics or TV based text display.
1	640x480 (80x40)	Text mode. Supports two colours per character row, with 8x12 characters. Suitable for console displays etc.
2	1024x768 (128x64)	Hi-res text mode. Supports two colours per character row, with 8x12 pixel characters. Suitable for console displays and text based GUIs on modern TFT displays of 15" or bigger.

The GPU can detect a monitor attached to the VGA port that supports 640x480 or 1024x768. If either of these modes is detected on startup the GPU selects the 640x480 text mode and puts the TV-output into sleep mode. Querying the supported modes will provide a value of 2 if a 1024x768 capable display has been detected or a 1 if the display is capable of only 640x480. If no display is found on the VGA port at startup then this means either there is none or it does not support plug-and-play detection. In this case the GPU reverts to the 512x480 tile based mode, the TV display is then driven with the same output so either display can be used as the primary output.

Once booted it is possible to switch to any other mode by sending a set mode command. This can be used for instance to switch to "graphical mode" where the fast 16x16 tile mode can be used to generate low colour images.

When a mode switch command is issued the GPU_READY bit in the status register is set. This bit is cleared again once the GPU has finished switching modes. Commands and data sent in this time may not be handled correctly so it is recommended not to send any.

Chapter 4

Keyboard Controller

4.1 Keyboard Controller Commands

You can set various keyboard parameters by writing to the KEY_DATA port. The basic format for these commands is an upper nibble specifying the command and some parameters in the lower nibble (allowing a total of 16 commands).

Command	Name	Description	Default
1100xxxx	KB.SET_TM	Sets the translation mode to xxxx, this is an internal setting that defines what bytes are sent to the Z80 on keypress. See Table 4.1 for details on implemented translations.	1 (ASCII)
1101000x	KB.SET_REL	If x is 1, all subsequent “normal keys” will send release codes (0xF0 followed by keycode) on release.	0
1110000x	KB.SET_CMD_REL	If x is 1, all subsequent control keys will send release codes (0xF0 followed by keycode on release.	1
11110xxx	KB.SET_LEDS	This command sets the LEDs (and internal state flags) to xxx. The MSb is CAPS Lock, the middle is NUM Lock and the LSb is SCROLL Lock.	010

4.2 Translation Modes

In the firmware implementation a relatively abstract lookup mechanism called `translate_key` is included. This allows arbitrary byte patterns to be sent on any keypress. The specifications of these translators is in the following sections, they can be selected with the `KB.SET_TM` command. Table 4.1 has a summary list of available translator functions.

4.3 ASCII Mode Codes

In ASCII Mode, printable characters will be sent, CAPS, SHIFT and NUM are taken into account before the byte is sent. In addition keys with direct mapping to ASCII codes will be sent as ASCII (Del = 127, Backspace = 8, Tab = 9, Return and Enter = 10). The only exceptions to this are two (UK) keyboard keys that don’t map to standard ASCII codes, and . The symbol is mapped

Number	Name	Description
0	None	No translation is done, LEDs are kept up to date and Pause/Break is filtered to send only one byte. F7 will also send 0xF7 instead of 0x83 other than that all key-press related codes will be sent as Set 2 scancodes.
1	ASCII	Keycodes that map directly will be sent as ASCII codes these are “normal keys” which won’t send release info by default. Command keys (e.g. shift, ctrl, cursor keys etc.) will send custom ndcodes which guarantee 1 byte identification. They will send release info by default.

Table 4.1: Translation Modes

to 0xA3, which matches the GPU’s internal font mapping, the symbol has no representation in the GPU and is replaced with a degrees symbol 0xB0. Neither of these codes are used in ndcodes to avoid confusion.

Other keys are sent as ndcodes which is a simple mapping which makes sure bit 7 is set for all control keys and assigns a single unique byte to each key. These are shown in the table below.

4.3.1 Control Keys

Mnemonic	Hex	Description
CC_ESC	0x80	Escape
CC_F1	0x81	F1
CC_F2	0x82	F2
CC_F3	0x83	F3
CC_F4	0x84	F4
CC_F5	0x85	F5
CC_F6	0x86	F6
CC_F7	0x87	F7
CC_F8	0x88	F8
CC_F9	0x89	F9
CC_F10	0x8A	F10
CC_F11	0x8B	F11
CC_F12	0x8C	F12
CC_PRINT	0x8D	Print Screen
CC_SCROLL	0x8E	Scroll Lock
CC_PAUSE	0x8F	Pause/Break
CC_NUM	0x90	Num Lock
CC_CAPS	0x91	Caps Lock
CC_LSHIFT	0x92	Left Shift
CC_RSHIFT	0x93	Right Shift
CC_CTRL	0x94	Left Control
CC_CTRLR	0x95	Right Control
CC_ALT	0x96	Left Alt
CC_ALTR	0x97	Right Alt
CC_GUI	0x98	Left GUI (Windows key)
CC_GUIR	0x99	Right GUI (Windows key)
CC_APPS	0x9A	Apps (Looks like a menu on most keyboards next to Windows Key)
CC_INS	0x9B	Insert
CC_HOME	0x9C	Home
CC_END	0x9D	End
CC_PGUP	0x9E	Page Up
CC_PGDN	0x9F	Page Down
CC_PSF	0xA0	Fake scan code, trapped internally.
CC_PSR	0xA1	Print Screen
0xA3 is symbol in our character set, don't use it here		
CC_LEFT	0xA4	Left cursor key
CC_RIGHT	0xA5	Right cursor key
CC_UP	0xA6	Up cursor key
CC_DOWN	0xA7	Down cursor key

4.3.2 Media Keys

CC_MNXT	0xA7	Media next
CC_MPRV	0xA8	Media previous
CC_MPP	0xA9	Media play/pause
CC_MSTP	0xAA	Media stop
CC_MMT	0xAB	Media mute
CC_MVU	0xAC	Media volume up
CC_MVD	0xAD	Media volume down
CC_MSL	0xAE	Media select
CC_MEM	0xAF	Media email
0xB0 is the degrees symbol in our character set		
CC_MCLC	0xB1	Media calculator
CC_MCMP	0xB2	Media my computer
CC_MSRCH	0xB3	Web search
CC_MHOME	0xB4	Web home
CC_MBCK	0xB5	Web back
CC_MFWD	0xB6	Web forward
CC_MWSP	0xB7	Web stop
CC_MRFSH	0xB8	Web refresh
CC_MFV	0xB9	Web favourites

4.3.3 ACPI Control Keys

CC_PWR	0xBA	Power
CC_SLP	0xBB	Sleep
CC_WK	0xBC	Wake

Chapter 5

PIC Pin Allocations

Figure 5.1 shows the functions of all the pins on the PIC. The crystal, power, programming and reset wiring have been left out for clarity. The high address latch is fed from the low address bus, with a straight through mapping (i.e. A0 latches into A8, A2 latches into A9 ... A7 latches into A15).

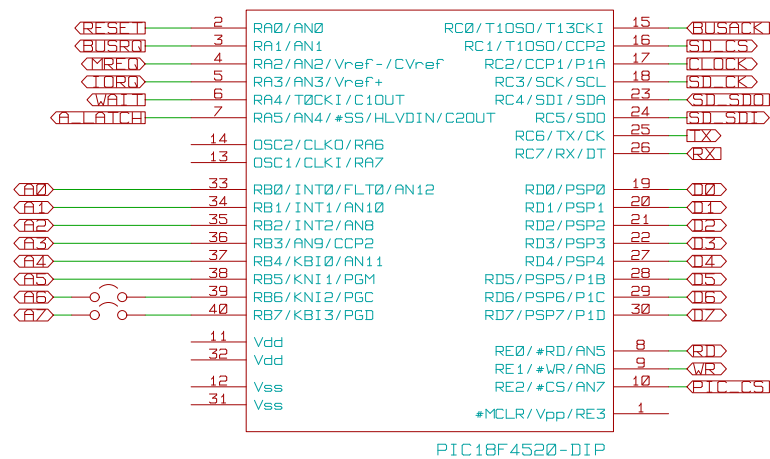


Figure 5.1: Pin allocations on the PIC.

Chapter 6

PIC Source Structure

The source code for the CPU supervisor PIC is split into a number of files (as of 17 Oct 2009) to aid development by compartmentalising code to make it easier to find, and also to make it easier to write “clean” code with minimal cross-references between blocks of different functions. The main source files are laid out below, as well as these assembly files there are 2 other files (as well as an include file and linker script from GPUUtils); the Makefile which automates building and programming the PIC and the portpins.inc file which declares the pinout of the PIC in handy names for bit-set/bit-test type operations.

6.1 Files

6.1.1 Main (main.asm)

The Main assembler file contains the glue functions that sticks all the other source files together. This sets up the interrupt address dispatching, the initialising code and the main loop.

External Functions

There are no externally callable functions in Main.

Internal Functions

Global Variables

Name	Description
MAIN_TEMP	A general purpose temporary register for use in the main thread (not interrupts). Use this only for short term storage where the value will be immediately consumed by a called function or it might be overwritten.

6.1.2 Serial (serial.asm)

The serial assembler file contains all the code relating to the RS232 debug port on the PIC. This code is used to allow a PC to control the Z80 CPU by giving access to DMA, with IO and Memory read and write commands. This source file contains all the functions from the basic send and receive up to the specific command handler functions. For details of the communications protocol and command codes see Chapter 7.

External Functions

Name	Description
<code>serial_init</code>	Initialisation routine for all hardware and software set up for the serial comms routines. Called by <code>init</code> in the main file when the PIC is started up.
<code>serial_rx_int</code>	The interrupt handling routine. Called from the main interrupt service in <code>main.asm</code> when a serial receive interrupt event occurs.
<code>serial_command_dispatch</code>	Command dispatcher, called from the main loop. This checks to see if there are any commands waiting to be run and directs execution accordingly.

Internal Functions

Global Variables

6.1.3 Z80 Bus (`host_bus.asm`)

The Z80 Bus assembler file contains all the functions for accessing the host CPU bus, e.g. writing to an IO port, or putting the Z80 into DMA slave mode. See the `sd_card.asm` section for the Slave device functions.

External Functions

Name	Description
<code>get_reset</code>	Call this function to put the Z80 into reset and set the PIC as the bus master.
<code>get_dma</code>	Call this function to put the Z80 into DMA mode and set the PIC as the bus master.
<code>get_slave</code>	Set the PIC to be a normal IO device on the bus. This releases control of the bus and releases the RESET and BUSRQ lines, it also enables interrupts on the PSP so that read/write from the PIC is handled by interrupt.
<code>ensure_master</code>	Call this to make sure that the Z80 is master, if it is this returns immediately, if not it puts the Z80 into DMA slave mode. This is designed to allow random access to peek at memory or peripherals without setting up a master session manually each time, but allows for a master session to be set up by first calling <code>get_reset</code> or <code>get_dma</code> .
<code>revert_master</code>	Call this after a call to <code>ensure_master</code> to return the bus to the previous state (e.g. PIC as slave etc.)
<code>io_read</code>	Read a single IO address, the address is taken from <code>HI_ADDR</code> and <code>LO_ADDR</code> , data is returned in <code>DREG</code> . No mode checking is performed when this is called so it is the calling routine's responsibility to check that the PIC is allowed to drive the bus.
<code>io_write</code>	As with <code>io_read</code> except that the content of <code>DREG</code> are written to the address provided.
<code>mem_read</code>	Identical to <code>io_read</code> except that it drives the memory bus instead.
<code>mem_write</code>	Identical to <code>io_write</code> except that it drives the memory bus instead.

Internal Functions

Global Variables

Name	Description
<code>HI_ADDR</code>	High byte of address for bus operations.
<code>LO_ADDR</code>	Low byte of address for bus operations.
<code>DREG</code>	Data buffer for bus operations.

6.1.4 SD Card Functions (`sd_card.asm`)

When the PIC is in peripheral mode it sits on the bus acting as a pass-through device for accessing an SD card in MMC (SPI) mode. This allows the Z80 to do simple byte operations to it even though the SD card interface is serial. The PIC does not do any of the file system handling or higher functions though. The main interface to this code is interrupt driven from the PSP peripheral on the PIC.

External Functions**Internal Functions****Global Variables****6.1.5 Z80 Boot (boot.asm)**

The Z80 boot file contains a few high level functions that are used to allow the system's boot ROM to be stored in the PIC's flash memory. This code is called at start-up to copy the boot ROM into system RAM. There is also support for sending the boot code to the host PC and for downloading a new boot code from the host PC within this source file.

External Functions

Name	Description
boot_init	Setup all the external functions for driving the Z80. This is mainly to do with the software-generated clock signal that is made by one of the PIC's PWM peripherals.
boot_load	Copies the ROM image from the top 8K of the PIC's Flash into the bottom 8K of the system memory map.
boot_update	Copies a 128 byte block from the RX buffer to the internal Flash.

Internal Functions**Global Variables****6.1.6 Boot ROM (rom.asm)**

This file has no functions, it contains a chunk of Z80 machine code that is used to boot the main CPU after it is copied to system RAM on boot. This file provides a convenient way to include the code in to the final hex file for the PIC.

External Functions

There are no external functions in this file.

Internal Functions

There are no internal functions in this file.

Global Variables

Chapter 7

Debug Comms Protocol

7.1 Buffer Locations

RX Buffer: 0x0100-0x01FF, Used through INDF0

TX Buffer: 0x0200-0x02FF, Used through INDF1

7.2 Packet Specifications

7.2.1 Host to Device Packet Definition

Name	Length (Bytes)	Description
Command	1	A byte in the range 0x00-0x1F (0-31) See the instruction table below for details.
Data Length	1	Length (in bytes) of the data payload of the packet. Maximum value is 253 so that the whole packet fits into the buffer.
Data	<i>data length</i>	Any byte string specified by the length. Can be zero length.
Checksum	1	An XOR checksum of all the bytes in the packet from the Command to the last byte of the data.

7.2.2 Device to Host Packet Definition

Name	Length (Bytes)	Description
Response	1	A Response code. Normally based on the command that this is responding to. See the response code details below.
Data Length	1	Length (in bytes) of the data payload of the response. Any value from 0-253 is valid, but depends on what the expected response requires.
Data	<i>data length</i>	The byte string being returned. This is not included for a simple acknowledge message.
Checksum	1	An XOR checksum of the response packet.

7.3 Response Codes

As was stated in the above table the response code is normally just the command code that the response belongs to. However there are several exceptions to this. The simplest is the error response. If an error occurs (e.g. the wrong number of parameters are provided for a command or the command does not do anything) an error packet is sent back to the host. The response code in this case is the command code with bit 6 set (i.e. a bitwise and with 0x40). Hence if command 0x0A encounters an error, the response code would be 0x4A.

In cases where the error could cause corruption of the command code or the code was invalid the PIC will never send an un-defined code, instead it responds with a special code with bit 7 set (0x80-0xFF). For example if the checksum fails, the error may be in the command, so it is not sent back to the PC, instead 0x80 is used. These 'special' responses are in the special column of the error codes table below.

7.4 Command Codes

There are 32 command codes numbered sequentially from 0 to 31. If a command is listed as 'Unused' then it will respond with an "Unused Command" error packet. The parameters are what is transmitted in the payload of the packet, in the order listed. Addresses are transmitted Big Endian (i.e. high byte first). The Mnemonic field is a simple text string representing the value for clarity of code, this is what is included in the seriallib Python library for example.

7.4.1 Summary

Code	Mnemonic	Description	Parameters
0	None	Reserved	
1	None	Unused	
2	RDMEM	Read Mem	Address High (1 Byte) Address Low (1 Byte)
3	WRMEM	Write Mem	Address High (1 Byte) Address Low (1 Byte) Data (1 Byte)
4	RDMEMBLK	Block Mem Read	Start Address High (1 Byte) Start Address Low (1 Byte) Length (1 Byte)
5	WRMEMBLK	Block Mem Write	Start Address High (1 Byte) Start Address Low (1 Byte) Data Block (<i>Length</i> -2 Bytes)
6	RDIO	Read IO	Address High (1 Byte) Address Low (1 Byte)
7	WRIO	Write IO	Address High (1 Byte) Address Low (1 Byte) Data Byte (1 Byte)
8	UPDBIOS	Update BIOS	Start Address High (1 Byte) Start Address Low (1 Byte) Data (128 Bytes)
9	DOCMD	Do Command	Command code (1 Byte)
10	None	Unused	
11	None	Unused	
12	None	Unused	
13	None	Unused	
14	None	Unused	
15	None	Unused	
16	None	Unused	
17	None	Unused	
18	None	Unused	
19	None	Unused	
20	None	Unused	
21	None	Unused	
22	None	Unused	
23	None	Unused	
24	None	Unused	
25	None	Unused	
26	None	Unused	
27	None	Unused	
28	None	Unused	
29	None	Unused	
30	None	Unused	
31	None	Unused	

7.4.2 Mem Block Write

The memory block write command writes a block of data to the main memory at the location specified in the command. The first data byte specifies the upper address byte for the start point, and the second is the lower address byte. The rest of the data bytes (the packet length field - 2) are then written to memory starting at this address, so a total of 251 bytes may be written in one go. The address is incremented after each write so the byte order for the frame is little endian, this is opposite to the address byte order.

7.4.3 BIOS Update

The Z80 BIOS is stored in the PIC's internal Flash memory and copied to the system at boot. The BIOS can be updated through the PIC's debug serial port without re-programming the whole PIC firmware. This is done in chunks of 128 bytes. There is no requirement to update all 8K of the BIOS image at the same time, nor do you have to do it in any particular order. Each 128 byte block must be aligned at a 128 byte offset (i.e. bits [6:0] of the address need to be 0). The address is the address within the BIOS itself, the PIC will automatically redirect writes to the appropriate part of Flash, so an address in the range 0x0000-0x1F80 is valid. On successful write of the block an ACK will be sent, if the parameters given are not acceptable various error messages may be produced (see below). If the write fails verification, a 0x0007 error message is sent. In this case simply try this block again as the BIOS storage space doesn't affect the PIC's operation in anyway the debug kernel cannot be damaged by a failed BIOS update. Operation of the Z80 if a reset is performed is likely to go wrong.

7.4.4 Do Command

The Do Command instruction is a special instruction which is used for executing simple no-data instructions, e.g. reset. The packet contains a single data byte which is the name of the instruction to execute (there are up to 256 instructions 0x00-0xFF). The response is simply the same packet, or an error code, none of the "Do" subset of commands can reply with any data.

Do Code	Mnemonic	Description
0	None	Reserved
1	DOGETRST	Pull Z80 Reset line low until further notice.
2	DOGETDMA	Put the Z80 into DMA slave mode until further notice.
3	DOGETSLA	Release the Z80 into normal run mode, make the PIC a slave device.
4	DORST	Issue a software Reset to the PIC. Resets the whole system, wiping RAM and re-bootin the Z80 from the stored ROM image.
Other	None	Not defined, will return Error 9.

7.5 Error Messages

The minimum length for an error packet is 2 Bytes. The first two bytes of the error packet are always an error code to indicate what the problem is. In some cases where more information is available this is included after the error number, for example if the wrong number of bytes were sent with the command the number of bytes expected is specified in the response packet. The error codes are specified below, in the error packet the high byte of the error code comes first (directly after the length field) followed by the low byte.

Code	Special	Description	Additional Data
0		No error	None
1	0x80	Checksum Error	The checksum calculated by the PIC (1 Byte)
2	0x81	Bad Command (greater than 31)	The code the PIC received.
3		Unused Command	
4		Wrong number of parameters	The correct number for this command.
5		The requested data is too big for one packet	
6		The start address for BIOS update was not within a valid range (max 0x1FFF)	
7		The offset for the BIOS was not aligned to the 128 byte packet size.	
8		There was a verification error for this BIOS update.	
9		Unknown do command.	The unknown command.

7.6 seriallib.py

`seriallib.py` is a Python library which automates a lot of the basic processing functions of the protocol. Include it in a project by placing a copy in your Python path, or in the project folder then adding the line `include seriallib`. Within the library are constants to save remembering the numerical values of commands e.g. typing `seriallib.DOCMD` will be treated as an integer of value 9 (the command code for a Do Command packet). The rest of the functionality is wrapped in classes.

7.6.1 Packet

The `Packet` class is a representation of the packet type which automates length calculation and checksum generation whilst maintaining the flexibility of a string. Specific bytes can be read from it as from a string using subscripting e.g. the length of the packet may be retrieved as a character type from a packet `pkt` by subscripting byte 1; `pkt[1]`. Other string like functions that work with the packet are `len()` which returns the total byte length of the packet (including command, length and checksum bytes, so the number is equal to the length field + 3). Calling `str()` on the packet will return the byte string in a binary form i.e. there may be non-printable characters within it. This is the default representation used if you `print` the packet. To gain a more useful insight into the contents of the packet, calling `repr()` on it will return the whole packet as 2 digit, zero padded hexadecimal values separated by spaces, so for example a Do Get Reset command would appear as `09 01 01 09`.

Assigning Packet Values

There are two ways to assemble a packet using the `Packet` class. The best way for assembling a packet to transmit is by calling the `set_command` and `set_data` methods. `set_command` expects an integer between 0 and 31 which it will validate and convert to a byte for transmission, this is ideal for use with the constants declared to make code more readable. The `set_data` command expects a string which does not need to be printable characters, this forms the data payload of the packet un-altered. If you pass a small integer (less than 256) to `set_data` it will convert it to a single character. This is useful for using the Do Command integer constants, if a larger number is given it will produce an error. Note that `set_data` does not append to existing data so you cannot build a packet a bit at a time using this method. You do not need to set the checksum or length values as these are calculated when needed.

The other method of generating a `Packet` is using the `set_string` method. This takes one argument which is a byte string that is interpreted as a packet. When you assign data this way the checksum and length fields are validated to ensure this was a valid packet, then the bytes are assigned to the internal storage. You can then alter the command or data with the `set_command` or `set_data` methods, or print the packet etc. Much more useful in the case of a received packet is the ability to get a readable error message from it. The `Packet` class knows all the error codes and what data is returned with that type of error and will wrap this information up in a hopefully descriptive message. This error description can be aquired with the `get_error` method. This returns a tuple, the first element is an integer giving the error code, the second element is a descriptive string. The error code matches the codes presented here, where 0 is no error and higher numbers correspond to specific communications or system events. A useful test for an error is to perform an if on element 0 of the response

```
if pkt.get_error()[0] > 0:
    print "There was an error"
    print "  Code %d: %s" % pkt.get_error()
```

Chapter 8

PIC BIOS Commands

After boot, the PIC becomes a slave device on the Z80 I/O bus. It can then be interrupted by either a serial command on the debug port from a host PC (see Chapter 7) or by a command from the Z80.

8.1 Protocol

The PIC uses a similar protocol to the Vinculum USB host to throttle access, there are two flags in the Z80's STATUS register; SD_TXE and SD_RXF. These two flags permit reading (SD_RXF) when low and writing (SD_TXE) when low. The PIC will never talk without first being requested for data so SD_RXF will only go low when a request has been completed and resulted in some data. There is a third bit in the STATUS register called SD_RDY, this is asserted (low) by the PIC when an SD card is attached and has been successfully initialised. Failure to check the state of this pin before requesting data from the SD card will almost certainly result in a timeout and garbage data.

8.2 Commands

Commands take the form of single byte values. There are only 64 commands higher values are masked so will execute whatever command is specified by the lower six bits. If the value is not mentioned below, that command is not used and will perform no operation.

Commands should be written command and then all bytes of argument in sequence. No acknowledgement is provided after individual bytes. If the bytes cause any processing time to be required the SD_TXE flag will be set high to prevent further writes until the PIC is ready for the next byte. Responses are made as quickly as possible and are signified by a lowering of SD_RXF (also SD_INT).

8.2.1 BIOS_RESET_CMD \$3F

This command takes no arguments and returns no data. On reception of this command the PIC will immediately do a full system reset (as if the front panel reset button had been pressed).

8.2.2 BIOS_READ_VAR_CMD \$0A

Takes one byte argument; the address to be read, returns one byte of data; the contents of requested address in the PIC's internal EEPROM memory. This is used to fetch BIOS parameters such as the boot order and system clock speed.

Defined registers:

BIOS_BOOT_ADDRESS, Address \$00, contains the BIOS boot order. BIOS_CLOCK_ADDRESS, Address \$01, indicates the system clock speed.

8.2.3 BIOS_WRITE_VAR_CMD \$0B

Takes two byte arguments; the address followed by the data. Works in the same way as BIOS_READ_VAR_CMD to write to the PIC's internal EEPROM memory. No response is made on the completion of this command.

8.2.4 SD_CARD_CID \$20

Fetch the Card ID info from attached SD card. This takes no bytes of arguments and returns 17 bytes of data. The first byte of data is a status message, the ascii character O (letter, not zero) indicates the command was successful. A response starting with a letter E indicates an error, in this case, the second byte of the response is the last command that was sent to the SD card, followed by the error code returned. The rest of the packet is padded with undefined data. A response beginning with the character T indicates there was a timeout whilst accessing the card. In this case the second byte is the command last sent to the SD card followed by the last response received.

8.2.5 SD_CARD_CSD \$21

Fetch the Card Specific Data from the SD card (mainly useful for determining the size and class; HCSD or standard capacity). Takes no arguments and returns 17 bytes of data. The return format is as specified above.

8.2.6 SD_CARD_READ_BLOCK \$22

Reads a 512 byte block of data from the SD card. Takes a 4 byte argument, for HCSD this is the block number to fetch. For standard capacity cards it is a byte address of the block to fetch. This command returns 515 bytes, the first is the response status, (okay, error, timeout; see above) the following 512 are the data and the final pair are the CRC supplied by the SD card. As with the previous two commands this command always returns the requested number of bytes regardless of whether the command fails or not.

Chapter 9

Z80 BIOS Routines

The following routines are built into the Z80 bios. If they are documented here then they should be safe to call from external code. Where this is not so, the restrictions are noted in the description. Functions will not alter registers (other than A and F) unless these registers are specified as inputs or outputs.

The sources is split into a number of files which are roughly split based on the hardware they interface to (e.g. UART, RTC etc.) or a specific duty within the software (e.g. maths). The filename and linenumber of the jumplabel are stated at the start of each function description.

9.1 Alphabetical List of Functions

Source	Function	Address	Description
bios.z8a:296	bios_disable_int	\$8036	disable interrupts flagged in byte in A
bios.z8a:279	bios_enable_int	\$8026	enable the interrupt indicated by register A
bios.z8a:248	bios_error_exit	\$8019	called on a fatal error after GPU is ready
bios.z8a:334	bios_load_var	\$8054	loads a byte from non-volatile BIOS memory
bios.z8a:387	bios_reset	\$808d	ask the PIC to do a full system reset
bios.z8a:315	bios_save_var	\$8045	saves a byte of data to non-volatile bios memory
bios.z8a:264	bios_set_interrupts	\$801e	set PIE and the local copy of interrupt flags
fat.z8a:30	fat_init	\$8974	read a FAT boot sector and setup the FAT driver
fat.z8a:476	fat_next_block	\$8bf3	fetch the next block in the current file/folder.
fat.z8a:531	fat_next_cluster	\$8c36	fetch the next cluster of the active file
fat.z8a:308	fat_select_cluster	\$8b0a	(re)initialise the FAT firmware pointing at a specific cluster.
gpu.z8a:47	gpu_cls	\$8099	Clear the active screen
gpu.z8a:182	gpu_dec	\$80f3	Print a decimal value to the screen, maximum of 32bit input
gpu.z8a:79	gpu_get_colour	\$80a7	Return the active colour from the GPU
gpu.z8a:212	gpu_hex	\$8108	Print a hex representation of a number
gpu.z8a:33	gpu_init	\$8094	Reset the GPU to its power-on state
gpu.z8a:63	gpu_set_colour	\$809e	Select the colour on the active screen
gpu.z8a:161	gpu_str	\$80e9	Print a null terminated string
maths.z8a:255	maths_add32	\$843b	Add maths_op_b to maths_op_a and store in maths_op_a
maths.z8a:192	maths_asl32	\$841d	shift maths_op_a one bit left and put a zero in the lsb
maths.z8a:215	maths_asln32	\$8425	shift maths_op_a n bits left and zero the new lsbs
maths.z8a:122	maths_asr32	\$83fb	32 bit arithmetic shift right
maths.z8a:148	maths_asrn32	\$8403	32 bit arithmetic shift right by n bits
maths.z8a:27	maths_bcd_to_bin	\$83b7	convert a single byte from BCD to binary (in a)
maths.z8a:628	maths_bin_to_bcd	\$8605	convert a 32/16
maths.z8a:389	maths_div32	\$84de	calculate the div
maths.z8a:48	maths_mod	\$83c9	calculate a/b - returns b unaltered, a = a % b, c = a//b
maths.z8a:321	maths_sub32	\$848e	do a 32 bit subtraction maths_op_a - maths_op_b
maths.z8a:594	maths_test_z32	\$85ed	test a 32 bit value for zero
maths.z8a:575	maths_test_z64	\$85e4	test a 64 bit value for zero
uart.z8a:53	uart_write	\$8169	Writes the contents of A to the UART
usb.z8a:32	usb_get_byte	\$8d1c	read a byte from the Vinculum chip.
usb.z8a:48	usb_write_byte	\$8d25	write a byte to the Vinculum chip

9.1.1 bios_disable_int

bios.z8a:296 - \$8036

disable interrupts flagged in byte in A

No Documentation

9.1.2 bios_enable_int

bios.z8a:279 - \$8026

enable the interrupt indicated by register A

Description:

Register A contains a bit mask with a 1 in every interrupt to enable. This is carried out and the updated value recorded in system.interrupts. The interrupt enable is set at the end so don't call from interrupts.

9.1.3 bios_error_exit

bios.z8a:248 - \$8019

called on a fatal error after GPU is ready

Description:

This function is called by BIOS functions on errors, but only after the GPU has been started. When called **HL** contains a pointer to a string with some information about the error, this is printed then the system is halted.

9.1.4 bios_load_var

bios.z8a:334 - \$8054

loads a byte from non-volatile BIOS memory

Description:

A is loaded with a byte fetched from location **B** from EEPROM memory.

9.1.5 bios_reset

bios.z8a:387 - \$808d

ask the PIC to do a full system reset

Description:

In some cases it is useful to be able to reboot (after setting BIOS parameters for example.) In these cases it's important that all the support circuits get reset as well to enable proper boot, issuing a command to the supervisor PIC can tell it to do a full system-reset.

9.1.6 bios_save_var

bios.z8a:315 - \$8045

saves a byte of data to non-volatile bios memory

Description:

The contents of **A** are saved to the non-volatile BIOS setting storage in one of 256 locations as indicated by the contents of **B**.

9.1.7 bios_set_interrupts

bios.z8a:264 - \$801e

set PIE and the local copy of interrupt flags

Description:

Because the Peripheral Interrupt Enable register is write-only, a copy of the current flag settings is stored in memory, by using this function both of the locations are kept up to date. This is thread-safe as it blocks interrupts to ensure both locations are kept in sync so it is safe to alter the system_interrupts value in memory from interrupts, don't call directly as it enables interrupts. The value to set is in **A**.

9.1.8 fat_init

fat.z8a:30 - \$8974

read a FAT boot sector and setup the FAT driver

Description:

sd.block should be set pointing to the start of the FAT partition before calling this function. It should be called before any other FAT subroutines.

9.1.9 fat_next_block

fat.z8a:476 - \$8bf3

fetch the next block in the current file/folder.

Description:

Fetches the next block of the current file or folder. On return Carry is set if the fetch failed. If the fat_error register is clear then it just means the end of the file was reached.

9.1.10 fat_next_cluster

fat.z8a:531 - \$8c36

fetch the next cluster of the active file

Description:

This function finds the number of the next cluster in the active file/folder if there is one and calls fat_select_cluster to initialise that cluster. Returns directly with carry flag set and FAT_ERROR_NONE if there are no more clusters in the current file. Otherwise the return is from fat_select_cluster.

9.1.11 fat_select_cluster

fat.z8a:308 - \$8b0a

(re)initialise the FAT firmware pointing at a specific cluster.

Description:

Called after an update to `fat_cluster`. Points `fat_block` to the start of the current cluster and calculates `fat_last_block` for this cluster. Also performs error checking on the sector number, if the number is 0 it returns with carry set but no error in `sd_error`, the same behaviour is used for cluster numbers higher than the highest available. This works for FAT entry checking. On FAT16 requesting Sector #1 will return the first block of the root directory. On FAT32 this will fail like requesting 0 or high numbers.

9.1.12 gpu_cls

gpu.z8a:47 - \$8099

Clear the active screen

Description:

This sets all pixels in the active screen to the currently selected background colour. Reading any cursor location will return \$20 (the ASCII code for SPACE). Register **A** is used, but no arguments or return values are associated with this function.

9.1.13 gpu_dec

gpu.z8a:182 - \$80f3

Print a decimal value to the screen, maximum of 32bit input

Description:

This prints a decimal representation (in ASCII characters 0-9) of a numerical value to the screen. The number should be a normal binary number (to print BCD, call `gpu_hex`). The source is not affected, but the maths registers are used for the conversion to BCD. HL is a pointer to the source number, **B** contains the number of ASCII characters (counted from the least significant up) to print to the screen, **A** contains a number of bytes long the number to convert is (can be 1, 2, 3 or 4). If **A** is zero or 5 or more then 32bit is assumed.

9.1.14 gpu_get_colour

gpu.z8a:79 - \$80a7

Return the active colour from the GPU

Description:

This command queries the GPU to find out what the active colour is. This colour index is returned in **A** and must be in the range 0-7. The actual colour this represents depends on the colour map of the GPU at the time of the request.

9.1.15 gpu_hex

gpu.z8a:212 - \$8108

Print a hex representation of a number

Description:

This function sends ASCII characters representing 4 bit values to the GPU to display binary values from memory. **HL** contains a pointer to the location in memory of the bytes to represent, **A** contains a number of Hex digits to print. When called the function starts printing nibbles from **A** above **HL** and moves down. **HL** is not altered by this function, neither is the memory that is printed. This function can also be used to print BCD values directly.

9.1.16 gpu_init

gpu.z8a:33 - \$8094

Reset the GPU to its power-on state

Description:

On start up the GPU displays a blank screen on both VGA and TV monitors. It is in a 16x32 tile-based mode with a default colour table. Running this command orders the GPU to return to this state. Register **A** is used, but no arguments or return values are associated with this function.

9.1.17 gpu_set_colour

gpu.z8a:63 - \$809e

Select the colour on the active screen

Description:

There are 8 colours available at any time in the GPU. Each colour is a pair of foreground and background values selected from a possible range of 64 visible colours. This command selects which of the eight foreground/background colour pairs from the current colour map will be used when printing characters to the active screen.

The colour to use is passed in **A** and is masked to a value 0-7.

9.1.18 gpu_str

gpu.z8a:161 - \$80e9

Print a null terminated string

Description:

HL is a pointer to the first byte of a null-terminated string. The bytes are sent un-altered to the data port of the GPU until a null is found which causes the function to return. **A** is altered during this function, **HL** is incremented to the end of the string, no other registers or memory locations will be written.

9.1.19 maths_add32

maths.z8a:255 - \$843b

Add maths

.op

.b to maths

.op

.a and store in maths

_op
_a

Description:

Performs a 32 bit addition of a value in `maths_op_a` and `maths_op_b`. The result is stored in `maths_op_a` by default, to use `maths_op_b` or `maths_op_c` as the target call `maths_add32_b` or `maths_add32_c`. No registers are affected by this call. `maths_op_a + maths_op_b => maths_op_x` (depending on call) Flags: `maths_flags[C]` is set if the addition resulted in overflow, reset otherwise `maths_flags[Z]` is set if the result of the addition was zero, reset otherwise

9.1.20 `maths_asl32`

maths.z8a:192 - \$841d

shift maths

_op

_a one bit left and put a zero in the lsb

Description:

The contents of `maths_op_a` are shifted one bit to the left (upwards) and a zero is placed in the least significant bit. `[C] <- [31 <- 0] <- 0` Flags: `maths_flags[Z]` is set if the result is zero, reset otherwise `maths_flags[C]` is set if the most significant bit was 1 before the shift, reset otherwise The contents of internal registers are not affected by this function.

9.1.21 `maths_asln32`

maths.z8a:215 - \$8425

shift maths

_op

_a n bits left and zero the new lsbs

Description:

The contents of `maths_op_a` are shifted n bit to the left (upwards) based on the value of **A** and zeros are placed in the least significant bits. `[C] <n- [31 <n- 0] <n- 0` Flags: `maths_flags[Z]` is set if the result is zero, reset otherwise `maths_flags[C]` is set if the last bit shifted out was 1 reset otherwise The contents of internal registers are not affected by this function, **A** is not preserved however.

9.1.22 `maths_asr32`

maths.z8a:122 - \$83fb

32 bit arithmetic shift right

Description:

The contents of `maths_op_a` are shifted one bit to the right. The most significant bit is left unaltered (so the sign of the number is not changed). `[31] -> [31 -> 0] -> [C]` Flags: `maths_flags[Z]` is set if the result is zero, reset otherwise `maths_flags[C]` is set if the lsb was 1 before the operation, reset otherwise None of the CPU registers are affected by this function.

9.1.23 maths_asrn32

maths.z8a:148 - \$8403

32 bit arithmetic shift right by n bits

Description:

The contents of maths_op_a are shifted a number of bits to the right. The number of bits to shift is specified by the contents of **A** (the value is masked to 5 bits to provide a maximum of 31 bit shift). The most significant bit is propagated down each shift (so the sign of the number is not changed). [31] -n>[31 ->0] -n>[C] Flags: maths_flags[Z] is set if the result is zero, reset otherwise maths_flags[C] is set if the last bit shifted out of bit 0 was 1, reset otherwise The contents of the accumulator are not preserved by this operation, other registers are unaffected.

9.1.24 maths_bcd_to_bin

maths.z8a:27 - \$83b7

convert a single byte from BCD to binary (in a)

No Documentation

9.1.25 maths_bin_to_bcd

maths.z8a:628 - \$8605

convert a 322416

Description:

HL points to a value up to 4 bytes long to be converted, it is not altered so it is safe to point at values in ROM or that will be used later. **HL** is the least significant byte, **A** contains the number of bytes to copy (1, 2 3 or 4) these bytes are copied to maths_op_b with higher bytes set to zero. The result of the conversion is stored in maths_op_c and may be up to 5 bytes long.

9.1.26 maths_div32

maths.z8a:389 - \$84de

calculate the div

Description:

The contents of maths_op_a are divided by the contents of maths_op_b. The quotient is placed in the lower 32 bits of maths_op_c, the remainder is left in maths_op_a. On return carry is set if it was an illegal (division by zero operation).

9.1.27 maths_mod

maths.z8a:48 - \$83c9

**calculate a/b - returns b unaltered, a = a
% b, c = a//b**

No Documentation**9.1.28 maths_sub32***maths.z8a:321 - \$848e***do a 32 bit subtraction maths****_op****_a - maths****_op****_b****No Documentation****9.1.29 maths_test_z32***maths.z8a:594 - \$85ed***test a 32 bit value for zero****Description:**

HL points to the least significant byte of a 32 bit value to be tested. If the value is zero bit 1 (second least significant) of **A** is set, otherwise this bit is cleared. This matches the **Z** bit position in the maths_flags register so loading **A** with the contents of flags before a call and writing it back after will update the flags register. **HL** is left pointing 4 bytes above where it started, no other registers are affected.

9.1.30 maths_test_z64*maths.z8a:575 - \$85e4***test a 64 bit value for zero****Description:**

Works identically to maths_test_z32 but tests a total of 8 bytes starting at location **HL**. Used in testing results of 32x32 multiply etc.

9.1.31 uart_write*uart.z8a:53 - \$8169***Writes the contents of A to the UART****Description:**

UART write checks the status of the UART and sends once the buffer is empty. This is an indefinitely blocking call, however the UART clock can't be stopped in this system so it should always complete.

9.1.32 usb_get_byte*usb.z8a:32 - \$8d1c***read a byte from the Vinculum chip.**

Description:

This call fetches a single byte from the Vinculum chip. It obeys the status flag USB_RXF to avoid reading bytes before they are ready. Note that it will block indefinitely for bytes to be ready so call with caution.

9.1.33 usb_write_byte

usb.z8a:48 - \$8d25

write a byte to the Vinculum chip

Description:

Writes a single byte (from **A**) to the USB host chip. The USB_TXE flag is checked to make sure that the Vinculum is ready to accept bytes. If there are bytes to be read these are read and discarded as the output buffer must be empty before you try writing.

9.2 Alphabetical List of Variables

Source	Function	Address	Description
gpu.z8a:285	gpu_hex_lut	\$8142	16 byte lookup to convert one 4 bit value to an ASCII character
maths.z8a:804	maths_flags	\$86a9	flag register holds the flags from the last maths routine
maths.z8a:783	maths_op_a	\$8699	32 bit operator for maths routines
maths.z8a:789	maths_op_b	\$869d	32 bit operator for maths routines
maths.z8a:795	maths_op_c	\$86a1	64 bit result register for maths routines
bios.z8a:111	ram_top	\$8000	A pointer to the last byte in RAM
bios.z8a:237	stack	\$9fff	The bottom of (highest address used by) the stack
bios.z8a:197	system_boot_device	\$8017	Set by the BIOS bootloader to indicate where the boot image in use was found
bios.z8a:225	system_boot_file	\$8018	Set by the bootloader to the path to the currently running program file loaded at boot
bios.z8a:163	system_boot_order	\$8015	The order in which devices are searched for a boot image
bios.z8a:129	system_clock	\$8004	The speed of the Z80 clock
bios.z8a:141	system_clock_speeds	\$8005	Lookup table to convert clock setting to Hz
bios.z8a:182	system_filesystem	\$8016	indicates the filesystem type if boot was from SD card

9.2.1 gpu_hex_lut

gpu.z8a:285 - \$8142

16 byte lookup to convert one 4 bit value to an ASCII character

Description:

Since ASCII characters A-F do not follow 0-9 normally this is provided as a lookup to convert a 4 bit value to ASCII.

Access: Read-only

9.2.2 maths_flags

maths.z8a:804 - \$86a9

flag register holds the flags from the last maths routine

Description:

Bit 0: Carry, set when an add/subtract overflows. Bit 1: Zero, set when the result of the operation was zero. See commands for details about what happens to these flags.

9.2.3 maths_op_a

maths.z8a:783 - \$8699

32 bit operator for maths routines

No Documentation

9.2.4 maths_op_b

maths.z8a:789 - \$869d

32 bit operator for maths routines

No Documentation

9.2.5 maths_op_c

maths.z8a:795 - \$86a1

64 bit result register for maths routines

No Documentation

9.2.6 ram_top

bios.z8a:111 - \$8000

A pointer to the last byte in RAM

Description:

On boot the system runs a check through all memory addresses until it finds a location where RAM is present. This allows the system to be used with a less than full memory address space.

Access: Read-only

9.2.7 stack

bios.z8a:237 - \$9fff

The bottom of (highest address used by) the stack

Description:

The stack pointer is set to this location at boot and it grows downwards. At boot the bootup routines are stored between \$1000 and \$17FF, so the stack is limited to the 2K region above this. Once the bootloader has loaded an operating system, execution moves above the stack and it can extend down to the top of the variables area at \$1000 (now giving it a full 4K of room).

9.2.8 system_boot_device

bios.z8a:197 - \$8017

Set by the BIOS bootloader to indicate where the boot image in use was found

Description:

On boot, the bootloader stores a number here depending on the source of the boot image loaded.
0: No boot image found, (usually means the system is still in BIOS mode)

1: SD Card

2: USB device

Access: Read-only

9.2.9 system_boot_file

bios.z8a:225 - \$8018

Set by the bootloader to the path to the currently running program file loaded at boot

Description:

On boot, the system loads a file from the root directory of the first boot device (or subsequent if that fails based on BIOS settings) with the extension .z8b (Z80 boot image). The actual file name loaded is listed here as an 8.3 dos-style filename in a null terminated string. The filename is automatically padded on the right with spaces to fill 8 characters. The last four characters should always be ".Z8B".

Access: Read-only

9.2.10 system_boot_order

bios.z8a:163 - \$8015

The order in which devices are searched for a boot image

Description:

On boot this is read from the PIC's internal EEPROM memory and stored here. This setting is used by the bootloader to decide what order to search for a valid operating system boot image to load. This is a numerical value, the meaning of various values is described below. 0: Boot from SD card only

1: Boot from USB device only

2: Boot from SD as first choice, or USB if that fails

3: Boot from USB as first choice, or SD if that fails

Access: Read-only (except in BIOS setup)

9.2.11 system_clock

bios.z8a:129 - \$8004

The speed of the Z80 clock

Description:

On boot the clock speed is selected by a BIOS parameter held in the supervisor PIC's EEPROM memory. During the boot process this setting is queried and saved to memory. This can be used in timing critical applications, or to compensate for various clock speeds. The actual frequency in Hz can be looked up in `system_clock_speeds`.

0: 250 kHz
 1: 2.5 MHz
 2: 3.33 MHz
 3: 5.0 MHz

Access: Read-only

9.2.12 `system_clock_speeds`

bios.z8a:141 - \$8005

Lookup table to convert clock setting to Hz

Description:

This table contains one entry per possible clock setting (as stored in `system_clock`). The value of the clock speed in Hz for each setting is stored here as a 32bit integer.

Access: Read-only

9.2.13 `system_filesystem`

bios.z8a:182 - \$8016

indicates the filesystem type if boot was from SD card

Description:

The BIOS supports both FAT16 and FAT32 as boot filesystems, this field records the filesystem type to know which routines to use. \$06: FAT16
 \$0B: FAT32

Access: Read-only

9.3 Functions and Variables by Source File**9.3.1 `kb_commands.z8a`****Functions**

Source	Function	Address	Description
--------	----------	---------	-------------

Variables

Source	Variable	Address	Description
--------	----------	---------	-------------

9.3.2 usb.z8a

Functions

Source	Function	Address	Description
32	usb_get_byte	\$8d1c	read a byte from the Vinculum chip.
48	usb_write_byte	\$8d25	write a byte to the Vinculum chip

Variables

Source	Variable	Address	Description
--------	----------	---------	-------------

9.3.3 maths.z8a

Functions

Source	Function	Address	Description
255	maths_add32	\$843b	Add maths_op_b to maths_op_a and store in maths_op_a
192	maths_asl32	\$841d	shift maths_op_a one bit left and put a zero in the lsb
215	maths_asln32	\$8425	shift maths_op_a n bits left and zero the new lsbs
122	maths_asr32	\$83fb	32 bit arithmetic shift right
148	maths_asrn32	\$8403	32 bit arithmetic shift right by n bits
27	maths_bcd_to_bin	\$83b7	convert a single byte from BCD to binary (in a)
628	maths_bin_to_bcd	\$8605	convert a 32 ₂ 416
389	maths_div32	\$84de	calculate the div
48	maths_mod	\$83c9	calculate a/b - returns b unaltered, a = a % b, c = a//b
321	maths_sub32	\$848e	do a 32 bit subtraction maths_op_a - maths_op_b
594	maths_test_z32	\$85ed	test a 32 bit value for zero
575	maths_test_z64	\$85e4	test a 64 bit value for zero

Variables

Source	Variable	Address	Description
804	maths_flags	\$86a9	flag register holds the flags from the last maths routine
783	maths_op_a	\$8699	32 bit operator for maths routines
789	maths_op_b	\$869d	32 bit operator for maths routines
795	maths_op_c	\$86a1	64 bit result register for maths routines

9.3.4 sd_commands.z8a

Functions

Source	Function	Address	Description
--------	----------	---------	-------------

Variables

Source	Variable	Address	Description
--------	----------	---------	-------------

9.3.5 statics.z8a

Functions

Source	Function	Address	Description
--------	----------	---------	-------------

Variables

Source	Variable	Address	Description
--------	----------	---------	-------------

9.3.6 gpu.z8a

Functions

Source	Function	Address	Description
47	gpu_cls	\$8099	Clear the active screen
182	gpu_dec	\$80f3	Print a decimal value to the screen, maximum of 32bit input
79	gpu_get_colour	\$80a7	Return the active colour from the GPU
212	gpu_hex	\$8108	Print a hex representation of a number
33	gpu_init	\$8094	Reset the GPU to its power-on state
63	gpu_set_colour	\$809e	Select the colour on the active screen
161	gpu_str	\$80e9	Print a null terminated string

Variables

Source	Variable	Address	Description
285	gpu_hex_lut	\$8142	16 byte lookup to convert one 4 bit value to an ASCII character

9.3.7 uart.z8a

Functions

Source	Function	Address	Description
53	uart_write	\$8169	Writes the contents of A to the UART

Variables

Source	Variable	Address	Description
--------	----------	---------	-------------

9.3.8 fat.z8a

Functions

Source	Function	Address	Description
30	fat_init	\$8974	read a FAT boot sector and setup the FAT driver
476	fat_next_block	\$8bf3	fetch the next block in the current file/folder.
531	fat_next_cluster	\$8c36	fetch the next cluster of the active file
308	fat_select_cluster	\$8b0a	(re)initialise the FAT firmware pointing at a specific cluster.

Variables

Source	Variable	Address	Description
--------	----------	---------	-------------

9.3.9 sd.z8a

Functions

Source	Function	Address	Description
--------	----------	---------	-------------

Variables

Source	Variable	Address	Description
--------	----------	---------	-------------

9.3.10 rtc.z8a

Functions

Source	Function	Address	Description
--------	----------	---------	-------------

Variables

Source	Variable	Address	Description
--------	----------	---------	-------------

9.3.11 menu.z8a

Functions

Source	Function	Address	Description
--------	----------	---------	-------------

Variables

Source	Variable	Address	Description
--------	----------	---------	-------------

9.3.12 gpu_commands.z8a

Functions

Source	Function	Address	Description
--------	----------	---------	-------------

Variables

Source	Variable	Address	Description
--------	----------	---------	-------------

9.3.13 boot.z8a

Functions

Source	Function	Address	Description
--------	----------	---------	-------------

Variables

Source	Variable	Address	Description
--------	----------	---------	-------------

9.3.14 bios.z8a

Functions

Source	Function	Address	Description
296	bios_disable_int	\$8036	disable interrupts flagged in byte in A
279	bios_enable_int	\$8026	enable the interrupt indicated by register A
248	bios_error_exit	\$8019	called on a fatal error after GPU is ready
334	bios_load_var	\$8054	loads a byte from non-volatile BIOS memory
387	bios_reset	\$808d	ask the PIC to do a full system reset
315	bios_save_var	\$8045	saves a byte of data to non-volatile bios memory
264	bios_set_interrupts	\$801e	set PIE and the local copy of interrupt flags

Variables

Source	Variable	Address	Description
111	ram_top	\$8000	A pointer to the last byte in RAM
237	stack	\$9fff	The bottom of (highest address used by) the stack
197	system_boot_device	\$8017	Set by the BIOS bootloader to indicate where the boot image in use was found
225	system_boot_file	\$8018	Set by the bootloader to the path to the currently running program file loaded at boot
163	system_boot_order	\$8015	The order in which devices are searched for a boot image
129	system_clock	\$8004	The speed of the Z80 clock
141	system_clock_speeds	\$8005	Lookup table to convert clock setting to Hz
182	system_filesystem	\$8016	indicates the filesystem type if boot was from SD card

9.3.15 scancodes.z8a

Functions

Source	Function	Address	Description
--------	----------	---------	-------------

Variables

Source	Variable	Address	Description
--------	----------	---------	-------------

9.4 Functions by Memory Location

Source	Function	Address	Description
bios.z8a:248	bios_error_exit	\$8019	called on a fatal error after GPU is ready
bios.z8a:264	bios_set_interrupts	\$801e	set PIE and the local copy of interrupt flags
bios.z8a:279	bios_enable_int	\$8026	enable the interrupt indicated by register A
bios.z8a:296	bios_disable_int	\$8036	disable interrupts flagged in byte in A
bios.z8a:315	bios_save_var	\$8045	saves a byte of data to non-volatile bios memory
bios.z8a:334	bios_load_var	\$8054	loads a byte from non-volatile BIOS memory
bios.z8a:387	bios_reset	\$808d	ask the PIC to do a full system reset
gpu.z8a:33	gpu_init	\$8094	Reset the GPU to its power-on state
gpu.z8a:47	gpu_cls	\$8099	Clear the active screen
gpu.z8a:63	gpu_set_colour	\$809e	Select the colour on the active screen
gpu.z8a:79	gpu_get_colour	\$80a7	Return the active colour from the GPU
gpu.z8a:161	gpu_str	\$80e9	Print a null terminated string
gpu.z8a:182	gpu_dec	\$80f3	Print a decimal value to the screen, maximum of 32bit input
gpu.z8a:212	gpu_hex	\$8108	Print a hex representation of a number
uart.z8a:53	uart_write	\$8169	Writes the contents of A to the UART
maths.z8a:27	maths_bcd_to_bin	\$83b7	convert a single byte from BCD to binary (in a)
maths.z8a:48	maths_mod	\$83c9	calculate a/b - returns b unaltered, a = a % b, c = a//b
maths.z8a:122	maths_asr32	\$83fb	32 bit arithmetic shift right
maths.z8a:148	maths_asrn32	\$8403	32 bit arithmetic shift right by n bits
maths.z8a:192	maths_asl32	\$841d	shift maths_op_a one bit left and put a zero in the lsb
maths.z8a:215	maths_asln32	\$8425	shift maths_op_a n bits left and zero the new lsbs
maths.z8a:255	maths_add32	\$843b	Add maths_op_b to maths_op_a and store in maths_op_a
maths.z8a:321	maths_sub32	\$848e	do a 32 bit subtraction maths_op_a - maths_op_b
maths.z8a:389	maths_div32	\$84de	calculate the div
maths.z8a:575	maths_test_z64	\$85e4	test a 64 bit value for zero
maths.z8a:594	maths_test_z32	\$85ed	test a 32 bit value for zero
maths.z8a:628	maths_bin_to_bcd	\$8605	convert a 32 ₁₀ to 16 ₁₆
fat.z8a:30	fat_init	\$8974	read a FAT boot sector and setup the FAT driver
fat.z8a:308	fat_select_cluster	\$8b0a	(re)initialise the FAT firmware pointing at a specific cluster.
fat.z8a:476	fat_next_block	\$8bf3	fetch the next block in the current file/folder.
fat.z8a:531	fat_next_cluster	\$8c36	fetch the next cluster of the active file
usb.z8a:32	usb_get_byte	\$8d1c	read a byte from the Vinculum chip.
usb.z8a:48	usb_write_byte	\$8d25	write a byte to the Vinculum chip

9.5 Variables by Memory Location

Source	Variable	Address	Description
bios.z8a:111	ram_top	\$8000	A pointer to the last byte in RAM
bios.z8a:129	system_clock	\$8004	The speed of the Z80 clock
bios.z8a:141	system_clock_speeds	\$8005	Lookup table to convert clock setting to Hz
bios.z8a:163	system_boot_order	\$8015	The order in which devices are searched for a boot image
bios.z8a:182	system_filesystem	\$8016	indicates the filesystem type if boot was from SD card
bios.z8a:197	system_boot_device	\$8017	Set by the BIOS bootloader to indicate where the boot image in use was found
bios.z8a:225	system_boot_file	\$8018	Set by the bootloader to the path to the currently running program file loaded at boot
gpu.z8a:285	gpu_hex_lut	\$8142	16 byte lookup to convert one 4 bit value to an ASCII character
maths.z8a:783	maths_op_a	\$8699	32 bit operator for maths routines
maths.z8a:789	maths_op_b	\$869d	32 bit operator for maths routines
maths.z8a:795	maths_op_c	\$86a1	64 bit result register for maths routines
maths.z8a:804	maths_flags	\$86a9	flag register holds the flags from the last maths routine
bios.z8a:237	stack	\$9fff	The bottom of (highest address used by) the stack

Chapter 10

File Formats

The design of the OS for the Z80 project relies on a number of custom file formats. These have been designed to be simple but flexible.

10.1 Assembly files *.z8a

All the Z80 assembly code for the project is in files named with a .z8a extension. There is no special format to these files, however the use of the unique file extension is useful in highlighting the code correctly and distinguishing it from PIC assembly code, for example.

10.2 Bootable files *.z8b

Currently there is only one *.z8b file defined. This is the BOOT.Z8B default kernel image. The basic format for this file type is the same as other executables (see Section 10.5 Executable Files, for details), however to qualify as bootables, they must be compiled to run from address \$0000. The code will be loaded from this file up to address \$7FFF or the end of the file, whichever comes first. If a kernel grows beyond 32KB it is then responsible for loading the rest of the file into other pages of memory once it has been bootstrapped and is running itself. A BIOS variable is left by the bootloader to indicate which of the mass-storage devices the boot image was found on. This is sufficient currently to identify the BOOT.Z8B file to load any further data after boot.

Other files with this extension are reserved for future implementations of more flexible bootloaders.

10.3 Program files *.z8p

Standing for Z80 Program, the Z8P file is a standard executable format (see Section 10.5. These programs must be compiled to run from address \$8000. They will be loaded into memory until they reach address \$FFFF, or the end of the file, whichever comes first. If the program is larger than 32KB then it must request additional memory from the OS and add load these new pages with the rest of the code/data.

There are several entry points into a Program executable. These have yet to be defined but will include interrupt service routines etc.

10.4 Library files *.z8l

Library files are a special instance of executable files. Similar in purpose to .dll or .so files on Windows or Linux, they contain routines that may be used in many different programs that can be called into at run-time. They are loaded by the OS in response to library requests from programs. Only one copy of each library will ever be loaded at a time so all functions must be stateless i.e. they have no internal memory so that they can be called from multiple concurrent threads in interrupts etc.

Library file format details are not yet decided, other than the overall executable format.

10.5 Executable Files

All of the executable files share the following basic format:

Offset	Bytes	Description
0	1	Version key, for the Mark 2 project this will always be \$02. This ensures that the system does not try and run files for incompatible hardware.
1	2	Header length (HL), a 16 bit count from the start of the file to the first byte of the program code. This is always 3 or more to encompass the Version key and the header length itself. This flexible offset allows for any data up to 64KB in size to be stored in the header even if the software loading the program does not understand the header. Uses envisioned include author information, some sort of permissions structure or other general meta-data.
HL	Any	The actual program code starts in the first byte after the header.

The intention of using this format is to keep the code simple (read three bytes then perform a single seek operation to find the data) allowing fast loading of software. However there is room for version numbers or any other type of arbitrary meta data in the header even if the software loading this file doesn't understand what the header fields actually mean.