Z80 Project Mark 2: Documentation

Nathan Dumont

January 10, 2010

Contents

0.1 License

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.

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.

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	SDWR/SDRD	Access to the SD via the PIC while it is in slave mode.

Table 1.1: Z80 I/O port functions

1.1 RCSTA

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.

Chapter 2

Hardware Layout

2.1 UART

2.1.1 Jumpers

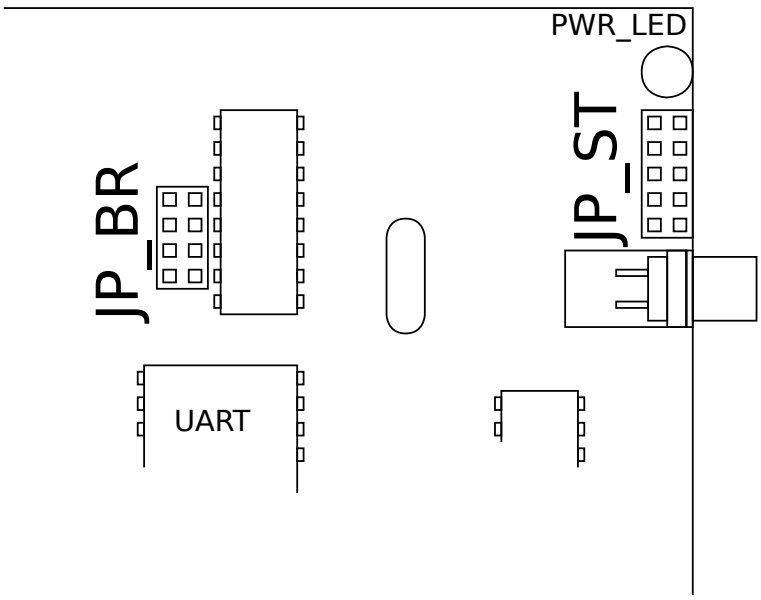


Figure 2.1: Jumper locations for UART settings

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!

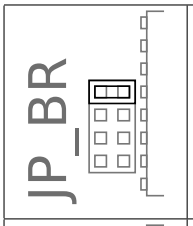
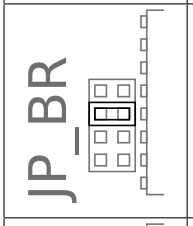
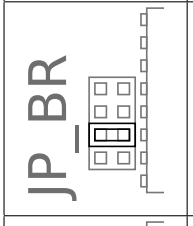
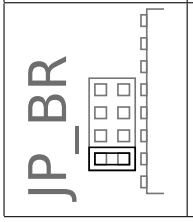
	4800
	9600
	2400
	19200

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.

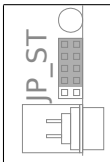
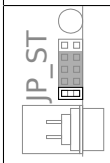
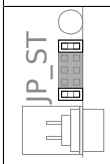
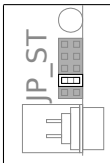
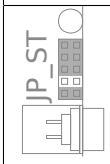
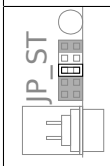
	No parity.
	Even parity.
	Odd parity.
	One stop bit.
	1.5 stop bits for character length of 5, 2 stop bits for all other lengths.
	5 bit character.
	6 bit character.
	7 bit character.
	8 bit character.

Table 2.2: UART configuration jumpers.

Chapter 3

PIC Pin Allocations

Figure 3.1 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).

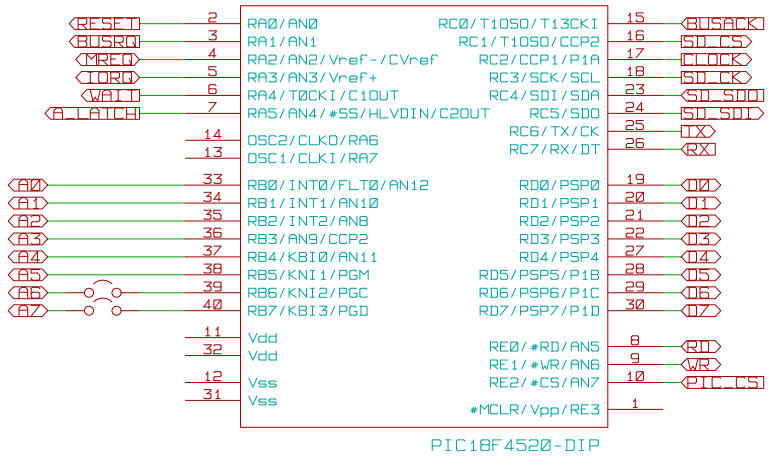


Figure 3.1: Pin allocations on the PIC.

Chapter 4

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 GPUtils); 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.

4.1 Files

4.1.1 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.

4.1.2 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 6.

External Functions

Name	Description
`serial_init`	Initialisation routine for all hardware and software set up for the serial comms routines. Called by `init` in the main file when the PIC is started up.
`serial_rx_int`	The interrupt handling routine. Called from the main interrupt service in main.asm when a serial receive interrupt event occurs.
`serial_command_dispatch`	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

4.1.3 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
`get_reset`	Call this function to put the Z80 into reset and set the PIC as the bus master.
`get_dma`	Call this function to put the Z80 into DMA mode and set the PIC as the bus master.
`get_slave`	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.
`ensure_master`	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 `get_reset` or `get_dma`.
`revert_master`	Call this after a call to `ensure_master` to return the bus to the previous state (e.g. PIC as slave etc.)
`io_read`	Read a single IO address, the address is taken from `HI_ADDR` and `LO_ADDR`, data is returned in `DREG`. 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.
`io_write`	As with `io_read` except that the content of `DREG` are written to the address provided.
`mem_read`	Identical to `io_read` except that it drives the memory bus instead.
`mem_write`	Identical to `io_write` except that it drives the memory bus instead.

Internal Functions

Global Variables

Name	Description
`HI_ADDR`	High byte of address for bus operations.
`LO_ADDR`	Low byte of address for bus operations.
`DREG`	Data buffer for bus operations.

4.1.4 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

4.1.5 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

4.1.6 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 5

Variables and Constants

Chapter 6

Debug Comms Protocol

6.1 Buffer Locations

RX Buffer: 0x0100-0x01FF, Used through INDF0
TX Buffer: 0x0200-0x02FF, Used through INDF1

6.2 Packet Specifications

6.2.1 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	*data length*	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.

6.2.2 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	*data length*	The byte string being returned. This is not included for a simple acknowledge message.
Checksum	1	An XOR checksum of the response packet.

6.3 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.

6.4 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.

6.4.1 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 (*Length*-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	

6.4.2 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.

6.4.3 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 PICs 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.

6.4.4 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.

6.5 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.

6.6 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 fucntionality is wrapped in classes.

6.6.1 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 retreived 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()
```