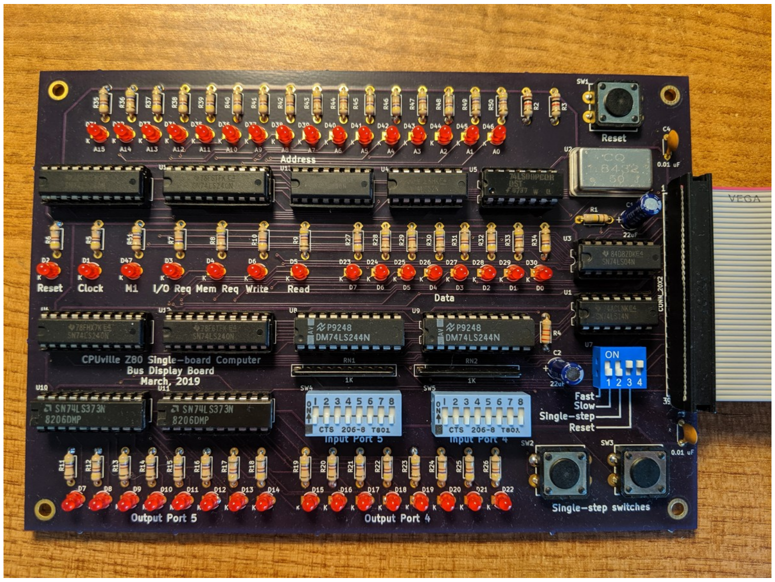
CPUville Single-board Z80 Computer Bus Display Instruction Manual

"The Slow Board"

by Donn Stewart

Table of Contents

Introduction



This is an accessory board for the CPUville Z80 Single-board computer. It provides a display of the address, data, and control system buses. It also has a slow clock of about 8 cycles per second, allowing you to examine the activity on these buses. This helps you understand the workings of this computer system, and also can be used to analyze hardware problems should they arise. In addition to the slow clock, there is a single-step clock. This is a bounceless toggle that allows you to provide single clock edges to the Z80, for an even closer look at the system activity on the buses.

Another feature of this kit is the addition of simple input and output ports (DIP switches and LEDs). These allow you to experiment with code in a way that is more straightforward than using the keyboard and display. The slow clock and simple ports make the Single-board computer system similar to the CPUville Original Z80 computer with its bus display, yet with the Single-board's 64K memory, serial interface and disk drive interface all operational.

Thank you for buying a CPUville kit. I hope you enjoy building and using it. If you have any questions, please contact me.

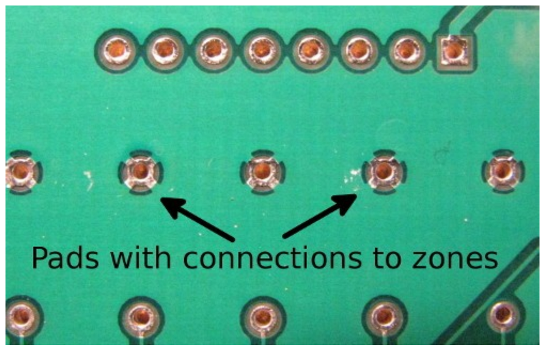
--Donn Stewart

Building the Single-board Bus Display

Soldering tips

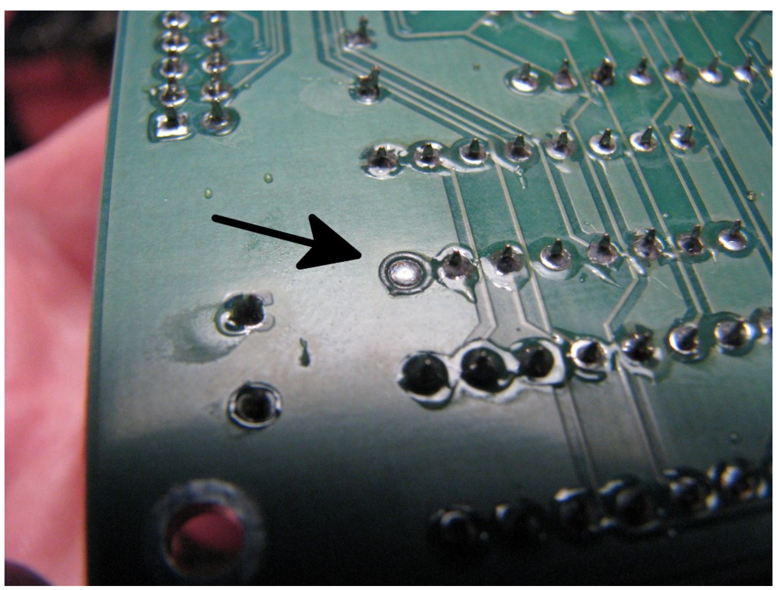
For a full explanation of soldering tips, see the CPUville Building and Soldering Tips document, available on the CPUville website kit instructions page.

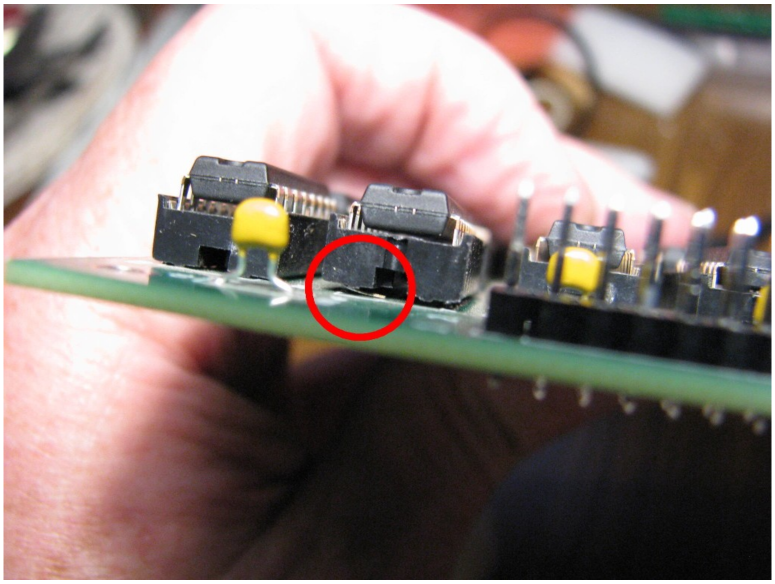
There are two main types of soldering errors. The first is failing to make a good connection to the pads of the ground zones:



These pads wick away the soldering iron heat, and may require higher wattage and/or more time to solder well.

The other main error is folding a pin under, but soldering the empty hole anyway, not realizing there is no pin sticking through:



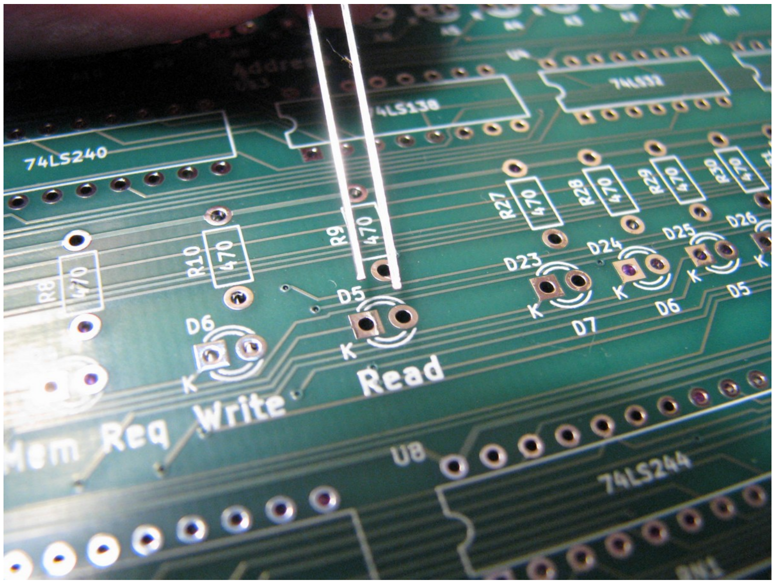


Both these errors can be corrected by careful attention during soldering, and by careful inspection of the board after it is finished.

Component placement

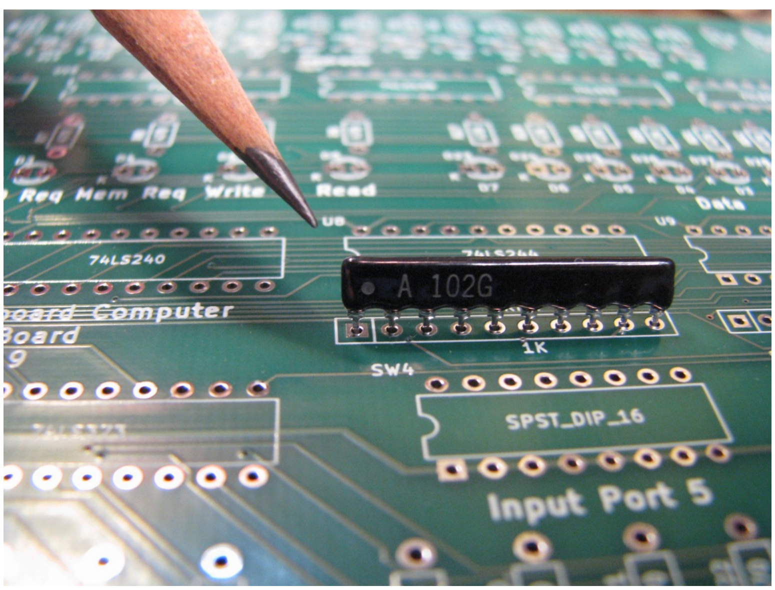
I usually place and solder the flattest components first, with the circuit board upside down sitting on the table top. So, I place them in this order: resistors, pushbuttons, resistor networks, oscillator, IC sockets, LEDs, ceramic capacitors, switches, and electrolytic capacitors.

Some of the components need to be soldered in the correct orientation. The LEDs go in with the short lead (cathode) inserted into the hole marked “K”:

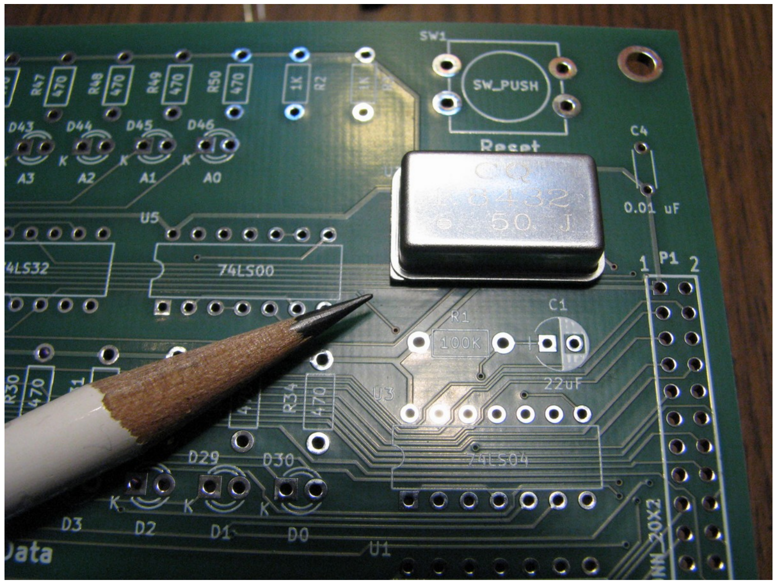


The plastic rim of the LED also has a flat edge on the cathode side.

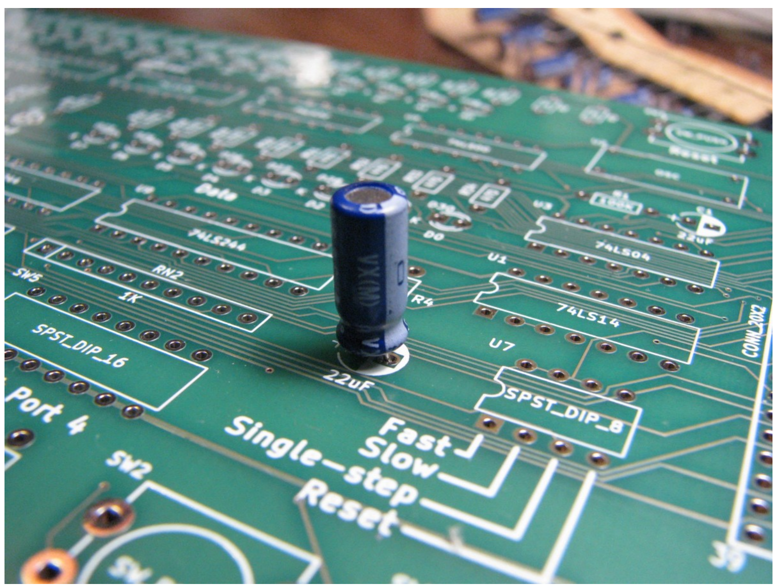
The resistor networks go in with the marked pin (pin 1) to the left, into the hole with the square drawn around it:



The oscillator goes in with the sharp corner at the lower left:



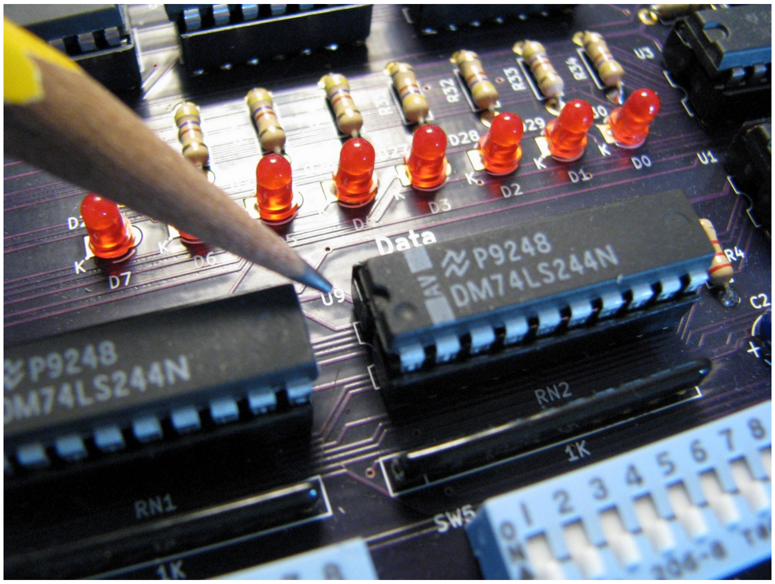
The electrolytic capacitors go in with the negative lead to the right:



The resistors, pushbutton switches, and ceramic capacitors do not need to be oriented. The DIP switches should be place so that “On” is up. The IC sockets do not need to be oriented, but they have a small cut-out on one end that can be placed on the left to match the cut-out on the ICs.

Once the components and sockets have been soldered in, carefully inspect each solder joint to make sure there are no solder bridges or cold joints, and that all the connections have a pin present. Use magnification to inspect the joints if possible. Note especially the condition of the solder connections to the ground plane zone. These require more heat and/or time to make, and are the most common source of soldering errors. Finally, brush off the back of the board to get rid of loose debris.

Plug the integrated circuits into their sockets. They need to be oriented with the small cut-out to the left:



Take extra care not to fold any pins under when plugging in the ICs. After each one is plugged in, look carefully down the length of the IC to be sure no pins are folded under, or out to the side.

Testing the bus display board

Once all the ICs have been inserted, and you are confident no pins are folded under or outward, it is time to test the board. The bus display LEDs are configured so that the buffer ICs will sink current through them, causing the LEDs to light up. The buffer ICs are inverting buffers. That is, when the input of a signal is low, the buffer output is high, and the LED will not light. When the input is high, the buffer output is low, and the buffer will "sink" current through the LED, and it will light. Unconnected buffer inputs usually assume a high state, so if you connect power to the board (+5V to pin 1 and ground to pin 40 on the bus display connector), leaving the signal pins unconnected, the LEDs will light. This is the easiest way to test the board. You can further test the board by grounding in turn the signal pins. If you ground a signal, its corresponding LED will go off.

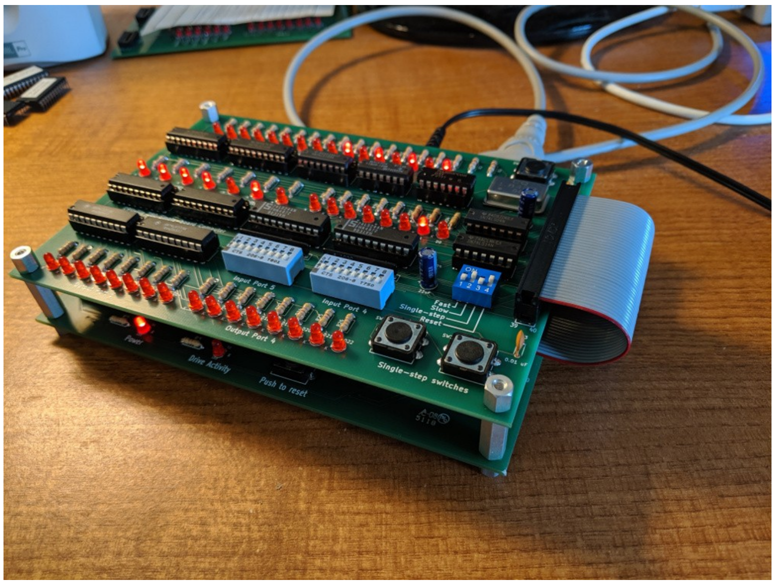
You can also do a quick check of the clock and reset circuits. With the fast clock selected, the Clock LED will be half-on, or dimly lit. With the slow clock selected, it will flash 8 to 10 times a second. With the single-step clock selected, it will turn on and off in response to pushing one or the other of the single-step switches. Also, if the Reset switch is on, pressing the Reset button will cause the Reset LED to go off, since Reset is an active-low signal.

There is no simple way to test the input/output ports without connecting the board to a working computer system.

Using the bus display

I assume you have a working Single-board Z80 computer system. Before you connect the bus display board, remove the external clock and reset jumpers from the computer board, because you will be using the clock and reset circuits on the bus display board.

With the power off, connect the bus display to the computer board using the 40-conductor connector. Make sure you have the connector plug holes lined up properly with the header pins. It is possible to place it off-center so that half the pins are not connected. Place the bus display board on top of the standoffs, and secure the board with the small standoff screws or nuts:



Connect the computer to the serial interface cable and activate your terminal emulation program as usual. Before you power-up the computer, select the fast clock by turning the fast clock switch on, leaving the slow clock and single-step switches off. Turn the reset switch on. Then power-up the computer.

You should see the ROM greeting message and the monitor prompt on the terminal display screen as usual. In addition, many of the bus display LEDs should light. You might notice some are bright, and others are dim. This is because some signals are cycling, and depending on the relative time spent in the high or low condition, the LEDs will vary in brightness. Pressing the Reset pushbutton will reset the computer, and cause the monitor greeting message and prompt to reappear.

Do a memory dump operation. You will see the LEDs flickering as the processor performs this action. Of course, the processor is going very fast, so it is not possible to see any detail in the bus activity.

Now we will test the simple input and output ports. Use the monitor `load` command to load the following bytes into RAM at location 0800h:

DB 04 D3 04 DB 05 D3 05 C3 00 08

These are the machine code bytes of a simple port reflector program. The assembly language for this program is:

```
Port_reflector    in a,(4)    ;Simple program to test ports
                  out (4),a
                  in a,(5)
                  out (5),a
                  jp Port_reflector
```

Once you have entered the machine code, run the program using the monitor `run` command, with the target address 0800h. Now, turn on some of the input port switches. The corresponding output port's LEDs should light. Turn the switches off, and the LEDs should turn off. This shows that the computer is reading and writing the ports correctly.

The displays

The displays show the system address bus, some control signals, and the data bus.

The data on the address bus is an output from the Z80. This address tells the system ROM, RAM or input/output ports which address location is to be written or read by the processor. For memory reads and writes, the full 16-bits of the address is used. For port reads and writes, only the lower 8-bits is used. One odd characteristic of the address bus is that during port operations, the port data (either being read or written) will appear on the upper 8-bits of the address bus. This is a non-documented "feature" of the Z80. Since the upper portion of the address bus is not used to select a port address, the presence of the port data there does not interfere with the port read or write function. I am not aware of any use for this, and it seems to have been deliberately made this way. Of interest, some manufacturer's Z80s do not do this.

The Z80 has many control inputs and outputs. I have selected a subset of these to put on the display. The Reset and Clock are inputs to the Z80, and the M1 (machine 1), I/O Req (input-output request), Mem Req (memory request), Write and Read signals are outputs from the Z80. The Reset LED shows the state of the CPU Reset input. When low, the Z80 is held in the reset condition. When Reset is released, and the Reset LED is on, the Z80 is in the run condition. The Clock LED displays the clock pulses that drive the Z80. The other Z80 outputs on the display are all active-low, that is, when asserted, the LED will be off. For example, if the Z80 is requesting a memory read, the Mem Req and Read LEDs will be off. The M1 LED shows when the Z80 is executing a "machine 1" cycle, which is an instruction fetch. On the slow or single-step clocks, by watching for M1, you can tell when a machine cycle is starting.

The data bus is bi-directional. If the processor is reading data, the bus is operating in one direction, and when writing data, in the other. The data is displayed in both cases, but the direction has to be inferred by looking at the control outputs. Since the data bus is bi-directional, all devices that place data on it must have “three-state” connections. The “third state” is a high-impedance state, like a disconnected wire. When not selected, the writing device must be in this third state. There are times that all devices that might write to the bus are in the third state. At these times, the display will show all the data LEDs on. You will notice this when you run the computer on the slow clock, that the default data bus display seems to be “all-on”. This is because the inputs to the bus display buffers that drive the LEDs, like most TTL inputs, will assume the high state if not connected. The third state is like being disconnected, so the inputs will assume the high state, and the LEDs will all light when no data is being written to the bus. Only when data is being written to the bus will some LEDs be off.

Using the slow clock

To use the slow clock, turn off the fast clock switch, and turn on the slow clock switch. You should see the Clock LED blinking about 8 times a second. Press and hold the Reset switch for a few seconds, then release it. The computer is now running the system monitor, but very slowly. After a few minutes you will see the monitor greeting message begin to appear on the terminal display.

Using the slow clock, it is possible to see in detail the activity in the microcomputer system. The control signals are active-low. So, for example, when the processor is doing a memory read, the Mem Req and Read LEDs will go off. Similarly, when the processor is writing to an output port, the I-O Req and Write LEDs will go off. You can tell a lot about what is happening by looking at the LEDs, but 8 cycles per second is still pretty fast for a human brain. To see maximum detail, you can take a video of the display, and look at the video frame-by-frame to see and understand all the activity. The following paragraphs explain some of what you will see.

The Z80 has a special M1 signal. This LED will go off when the processor is executing an M1 (“machine 1”) cycle. This is the first clock cycle in an instruction execution. The processor will display the address of the instruction to be read on the address bus, and set the Mem Req and Read signals low. This will cause the memory to place the 8-bit instruction opcode on the data bus. M1 lasts two clock cycles. At the end of the M1 cycle, the instruction opcode is placed into the instruction register inside the Z80.

The two cycles after M1 allows the processor time to interpret the opcode. During these cycles, the Z80 will perform a memory refresh. This is for systems with dynamic RAM. The Single-board Z80 computer uses static RAM, so the refresh is not needed, but the Z80 will perform it anyway. Since it performs the refresh during instruction interpretation, there is no performance penalty. During the refresh cycle, the Mem Req signal is activated (the Mem Req LED will go off) but neither the Read nor Write signals are activated. Systems with dynamic memory use the Mem Req signal, coupled with the Z80 Refresh signal (not shown on the display), and a refresh row address placed on the address bus to refresh their memory.

After the M1 and refresh cycles are finished, the processor will perform the instruction. There are more than one hundred instructions used by the Z80, so it is beyond the scope of this instruction manual to explain what is seen on the bus display. However, if you refer to the Z80 datasheet for your particular type of Z80, you will be able to understand a lot about how the Z80 interacts with its computer system.

We have been using the slow clock to run the system monitor. But what if you want to use the slow clock to examine the bus activity while running your own program? There are two ways this can be done. The first way is to enter the program in memory while on the fast clock, switch to the slow clock, and reset the computer. The reset will not affect the program in RAM. After the reset the monitor program will start running, and eventually get to the command prompt. This will take several minutes. At the prompt, you can enter commands one character at a time. So, if you have entered your program at location 0800h, switched to the slow clock and reset the computer, when it gets to the command prompt you can enter the `run` command. You need to press “r”, then wait until the “r” is displayed. Then count to 10. Then press “u”, and so on. When you have finished entering “run”, press Enter. Now you will have to wait a long time for the run command to print its message, and get to the point where you can enter the address. You enter the address one character at a time, as you did for the “run” command, then hit Enter. Now you have to wait another few minutes for the run command to convert the address character string into the binary address that the Z80 will jump to. Finally, your program will begin to run. You can tell it is running when you see the address of your program being displayed frequently on the address display. For the address 0800h, you will see the A11 LED lighting. All this will take 10 to 15 minutes.

Switching clocks while running

The other way to run your own program on the slow clock is to switch clocks while running. With most Z80s I have tested this works about half the time you try it. The other half, the Z80 is upset by the change and will freeze or jump to non-program memory locations. But, it is probably faster to switch clocks while running than to enter monitor commands on the slow clock, as outlined above. Switching clocks while running will not harm the Z80 or the hardware of your computer system.

The technique I have used that works the best is to change the clock switches using a pencil or a ballpoint pen with the point retracted. Using my finger for some reason decreases my chance of a successful switch.

Start by entering your program using the `load` command with the fast clock as before, then use the `run` command on the fast clock to start your program. Once you see your program is running, turn off the fast clock, then quickly turn on the slow clock. If you see cycling on the buses on the slow clock, you probably have had a successful switch. You can be sure by looking at the addresses being displayed, and at the control LEDs. For example, if you successfully switch clocks while running the port reflector program entered at address 0800h as shown above, you will see the A11 address LED lighting, and the I/O Req LED going off frequently. Place data on the input port switches, and after a short time, the data will be displayed on the output port LEDs.

If the clock switch upsets the Z80, the display will freeze or show that the Z80 is somewhere else in the address space, not running your program. In that case, switch to the fast clock, reset the computer, and enter the `run` command again. (Even if the Z80 goes crazy during the attempt to switch the clocks it is probable that the program in RAM is still there, so you will not need to reload your program). Then try the clock switch again.

Using the single-step clock

The reason clock switching is not always successful is that most Z80s do not tolerate the absence of clock input, especially prolonged periods of a low clock signal. I am not sure why this is the case. The datasheets for many Z80s will show the minimum clock frequency to be “D.C.” (direct current, meaning the Z80 should remain stable without receiving clock edges), but in practice they do not remain stable for long. You can see this if you try to single-step the Z80 you have. Select the single-step clock, then hold the Reset pushbutton while cycling the clock with the single-step pushbuttons. Once all the address and data LEDs have lit you know the processor has been reset. Now release the Reset pushbutton. If you continue to cycle quickly, at about 2 clicks per second, the processor will run. However, if you slow down or stop single-stepping, the display might not hold steady. Some address LEDs will light even when you are not stepping the clock. The control LEDs and data LEDs may also change. If you try to cycle again, the Z80 will not respond. By experimentation, it seems that many, but not all, Z80s will remain stable if you stop cycling with the clock level high. But if the clock input is low, the lights will begin to change, and the Z80 will be locked up. As long as you are careful to stop single-stepping with the clock high, you might be able to keep going. Single-stepping allows you to examine in detail the activity on the system buses one upgoing clock edge at a time. But if you are curious about what happens on a downgoing edge, the Z80 will probably not tolerate this.

There are exceptions to this. Some Z80s are capable of full single-stepping, and remain stable with the clock signal held high or low. The two I have tested are Zilog part number Z84C0006PEG and Toshiba part number TMPZ84C00AP-6. These are more expensive than the Z80s I provide with the kit, and to keep the kit price as low as possible I will continue to provide the less expensive processors. However, if there is strong interest in single-stepping with a processor that can be stopped with the clock low, I will offer a fully single-step capable Z80 as an option with the kit.

One advantage of having a fully single-step capable Z80 is that clock switching is much more reliable. Since the processor will remain stable without clock input, you can turn off the slow clock (leaving the processor with no clock input at all) and it will remain stable, waiting for you to turn on the slow clock or the single-step clock.

Using the v.15 ROM with the bus display

I have coded a new ROM for use with the Single-board Z80 computer with the bus display attached that combines features of the v.7 ROM shipped with the Original Z80 computer, and the v.8 ROM shipped with the single-board computer. Since it has features of both, I designated it v.15. A listing of this ROM can be found toward the end of this manual.

This ROM only works in a system with the bus display attached. At power up or reset, instead of jumping right to the system monitor program like the v.8 ROM, it reads an address from the input port switches and jumps to that address, like the v.7 ROM would do. This allowed me to put some small demo programs in the ROM that will run immediately after power up or reset, if the correct address is on the input port switches. This avoids the problem of loading demo programs using the system monitor and then trying to switch the clock speed while the system is running, as mentioned in the above section “Switching clocks while running”. You can simply put the address on the switches, set the clock to the speed you want, and press reset. The demo program will start to run after a few cycles (the instructions to read the switches and jump).

In addition to the code to read an address from the switches and jump to it, and the demo programs, the v.15 ROM has a full system monitor program that can perform all the functions of the monitor program in the v.8 ROM. The only difference is that I shortened some of the messages to make a little more room for the extra code of the demo programs. So, you can use the `dump`, `load`, `run`, `bload` and `cpm` commands just like the v.8 ROM. To enter the system monitor in the v.15 ROM at startup or reset, the monitor cold start address 0x0494 must be on the port switches.

These are the three demo programs:

Port reflector, address 0x0007 (binary 0000 0000 0000 0111)

This program gets a data byte from each input port, and displays it on each output port. On the slow clock you will note the use of the I/O_Req signal when the ports are read from or written to. Also, you might note that the data from every port read and write instruction appears on the upper 8 bits of the address bus in Zilog brand Z80s. This is an undocumented “feature” that does not interfere with the function of the port instructions, since the addresses of the ports are only 8-bits.

Simple counter, address 0x0012 (binary 0000 0000 0001 0010)

In this program, the Z80 increments the value in the A register and displays the result on output port 4. The output port display will go from 0 to 255 (binary 0000 0000 to 1111 1111) over and over again. It is useful to watch how the CPU operates the bus signals when the slow clock is on. With the fast clock the bus display and outputs are a blur.

Count to a million, address 0x001a (binary 0000 0000 0001 1010)

This program counts down 16 times by decrementing the A register, then increments the 16-bit register pair HL and displays the result on the output ports 5 and 4. The result is 16 x 65, 536 = 1,048,576 operations for a full cycling of the output. It is impressive to run this program with the slow clock, which seems to take forever to increment the output once, and compare that to the fast clock at 1.8432 MHz, which goes through the whole count in a second or two. This gives a visible demonstration of the speed of the computer.

Here is a list of the programs and the hex addresses for reference:

Port_reflector: 0007

Simple_Counter: 0012

Count_to_a_million: 001a

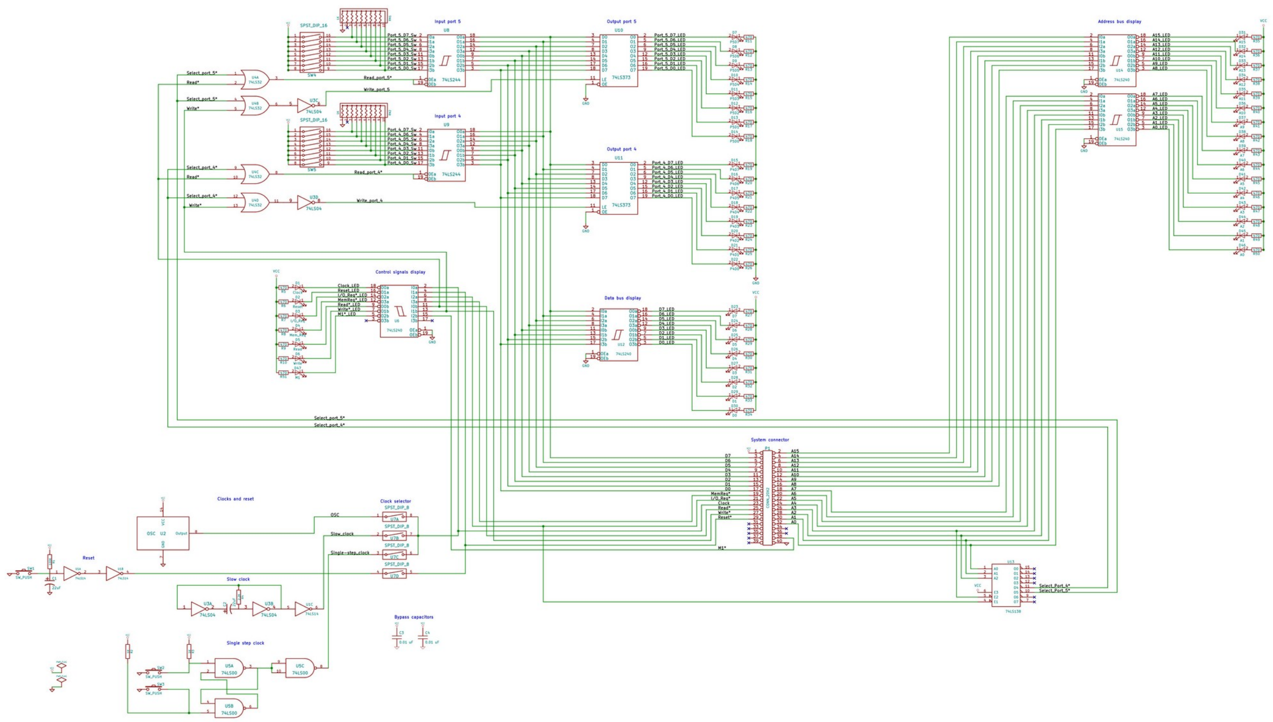
monitor_cold_start: 0494

monitor_warm_start: 04A0

The monitor warm start address is to be used to return to the ROM system monitor from any programs the user may have written, so that the computer will not need to be reset when the program is finished.

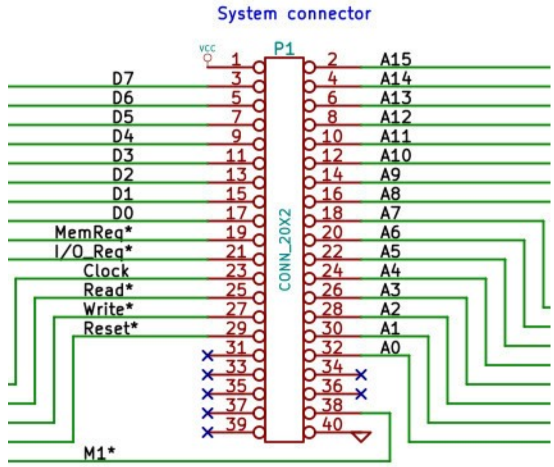
Schematics and explanations

Here is the whole schematic. Increase the view magnification to see the details. You can download the full resolution schematic from the CPUville website.



I have broken out the sections of the circuit and given explanations below.

System connector

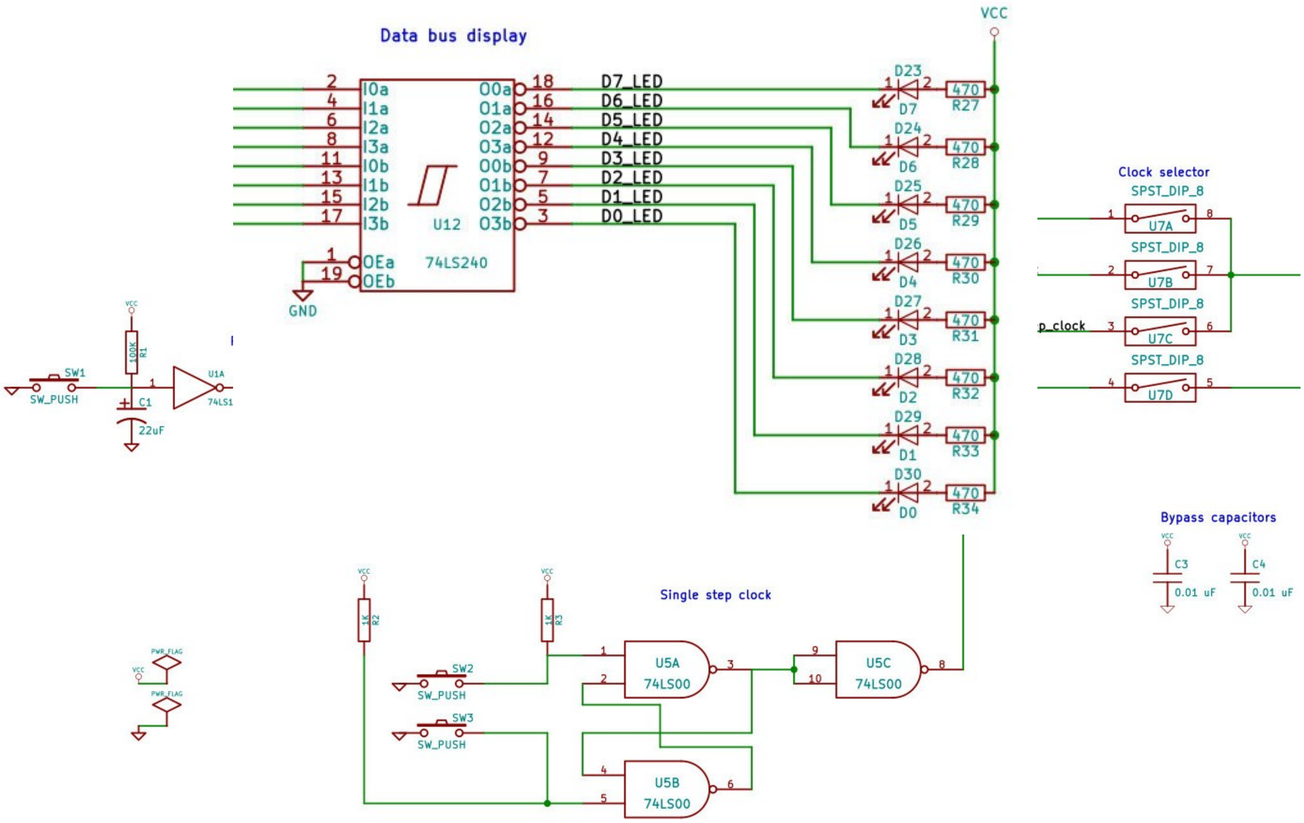


This is the bus display connector to the computer board. The Reset and Clock signals are inputs from the bus display to the computer board. The other signals are outputs from the computer board to the bus display. Please note that the computer system connector has other active signals that are not used by the bus display, here shown with “no connect” symbols (the blue x’s). These signals are shown on the Single-board Z80 computer schematic, available on the CPUville website.

Clocks and reset

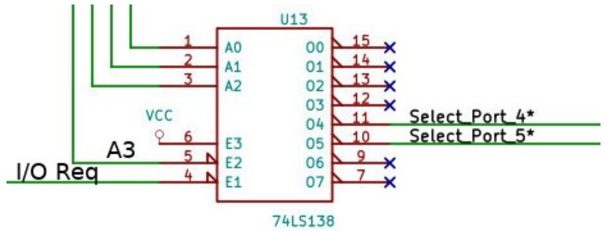
The fast clock (OSC) is a crystal square-wave oscillator that produces for this computer a 1.8432 MHz output. The slow clock is a resistor-capacitor oscillator connected to inverters that produce an approximately 8 Hz square-wave output. The single-step clock is a bounceless toggle switch that produces an up-down or down-up transition depending on which button is pressed. The reset circuit has a capacitor-resistor timer that holds the system in reset for about one second after power is applied, then releases. The reset pushbutton allows for resetting without disconnecting and reconnecting the power.

Display buffer and LEDs



This shows how the display LEDs are driven. Only the data bus display buffer and LEDs are shown, as an example, but the address and control displays use the same kind of circuit. The 74LS240 is an inverting buffer. The LEDs are arranged to that the buffer outputs will “sink” current when low, causing the LEDs to light. So, if a buffer input is high, the output will be low, and the LED will light because current can flow through the LED. If the input is low, the output will be high. In that case there is no voltage difference across the LED, so no current will flow, and the LED will be off.

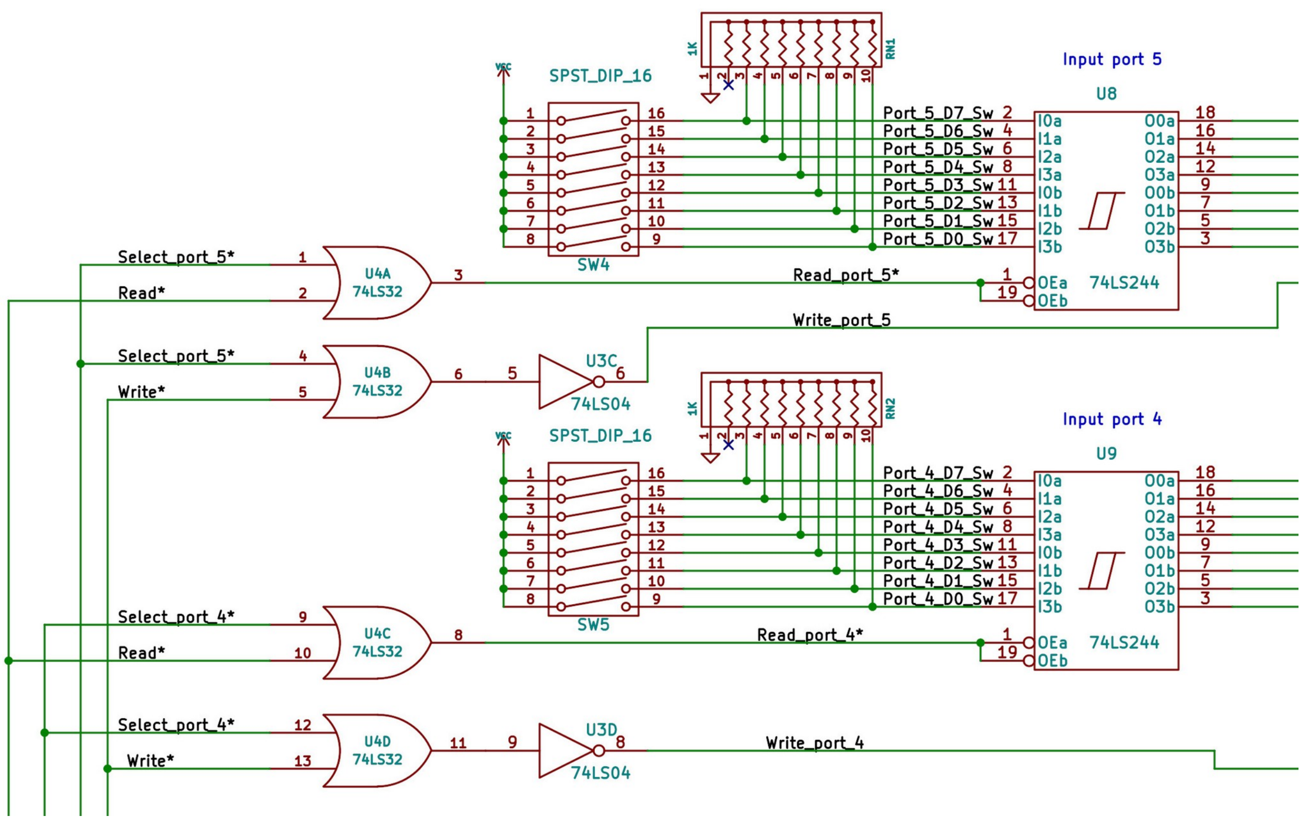
Port address decoder



The Single-board Z80 computer decodes ports 0 to 3, and 8 to 15 for its own use. Port values above 15 will “wrap around” and select these same ports again. However, ports 4 to 7 are not decoded on the computer board, allowing them to be used on an accessory add-on board, like this bus display board. This decoder is configured to select ports 4 and 5.

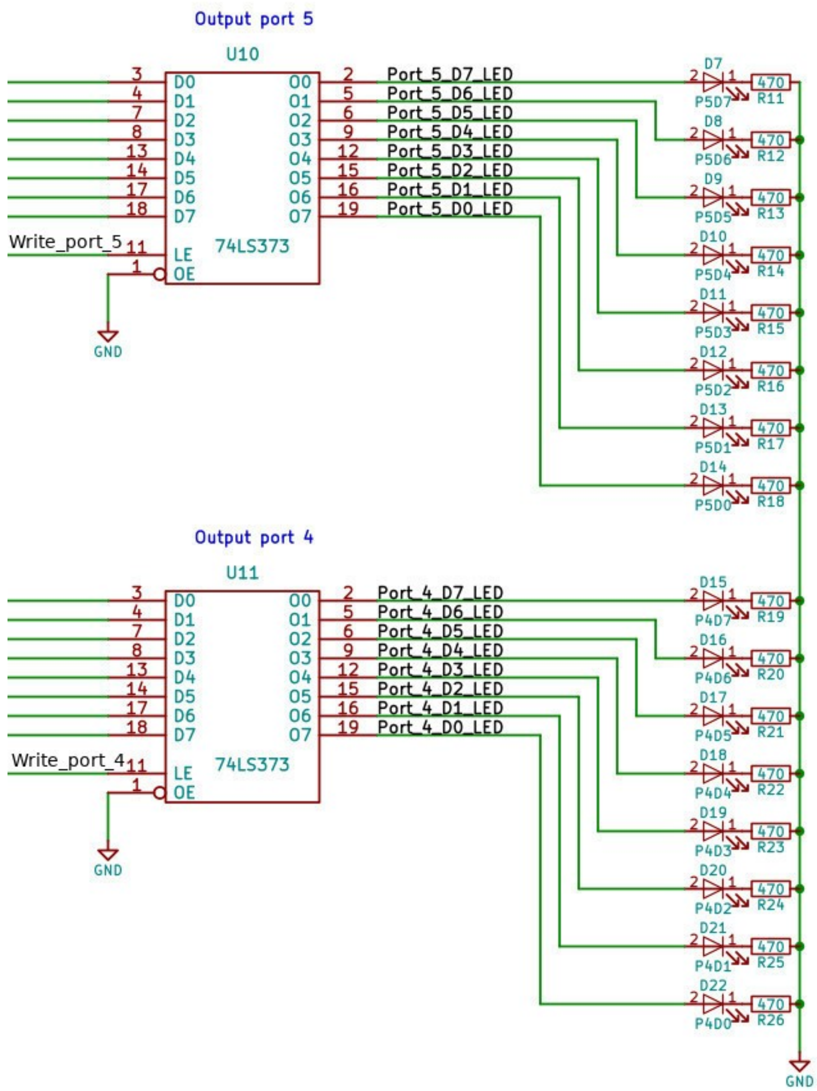
It selects one of eight outputs (causes it to go low) depending on the address input on A0 to A2. However, the decoder has three enable inputs, E1 to E3. The E3 input (active-high) is tied to VCC. The E1 (active-low) is connected to the I/O Req signal (active-low) and, E2 is connected to A3. The decoder outputs will only be asserted when there is a port request (E1 is low), and when A3 is 0, which is the case when requesting ports with addresses below 8.

Input ports



The input ports are non-inverting buffers with inputs controlled by the DIP switches, and the outputs connected to the data bus. The outputs are three-state, and will only be active when the output enable inputs Ea and Eb (active-low) are both asserted. This will happen if the port select (from the port decoder) and read signals are asserted together. The output port write signals go off the page to the output ports (see below).

Output ports



The output ports are transparent latches. The latch data inputs are connected to the data bus, and the outputs drive the LEDs. The write_port signals (active-high) from the input port schematic above are connected to the latch enable inputs. When the proper write_port signal is asserted, this will cause whatever data is on the data bus to be written into the latches, and the data will be displayed on the corresponding LEDs. When the latch enable is de-asserted, the data held in the latches will continued to be displayed.

Single-board bus display parts organizer

Capacitor, 0.01 uF 2	1.8432 MHz oscillator 1	4-position DIP switch 1	8-position DIP switch 2
74LS00 1	74LS04 1	74LS138 1	74LS14 1
74LS240 4	74LS244 2	74LS32 1	74LS373 2
40-pin header 1	Standoff, 0.25 inch M/F 4	Resistor network, 1K x 9 2	14-pin socket 4
16-pin socket 1	20-pin socket 8	Pushbutton 3	Resistor, 470 ohm Yellow-Violet-Brown 47
Resistor, 2.2K Red-Red-Red 1	Capacitor, 22 uF 2	LED 47	Resistor, 1K Brown-Black-Red 2
Resistor, 100K Brown-Black-Yellow 1			

Single-board bus display parts list

Part	PCB Reference	Number per unit	Jameco Part no.
Capacitor, 0.01 uF	C3,C4	2	15229
1.8432 MHz oscillator	U2	1	27879
4-position DIP switches	U7	1	38820
8-position DIP switches	SW4,SW5	2	38842
74LS00	U5	1	46252
74LS04	U3	1	46316
74LS138	U13	1	46607
74LS14	U1	1	46640
74LS240	U15,U14,U12,U6	4	47141
74LS244	U8,U9	2	47183
74LS32	U4	1	47466
74LS373	U11,U10	2	47600
40-pin header	P1	1	53532
Standoff 0.25 inch M/F		4	77586
Resistor network, 1K x 9	RN1,RN2	2	97877
14-pin socket		4	112214
16-pin socket		1	112222
20-pin socket		8	112248
Pushbutton	SW1,SW2,SW3	3	122973
Resistor, 470 ohm	R5 – R51	47	690785
Resistor, 1K	R2,R3	2	690865
Resistor, 2.2K	R4	1	690945
Resistor, 100K	R1	1	691340
Capacitor, 22 uF	C1,C2	2	1946295
LED	D1 – D47	47	2081932

40-conductor connector

v.15 ROM listing

```
# File 2K_ROM_15.asm
0000                  ;ROM monitor for single-board Z80 computer with bus display.
0000                  ;Mixed features of v.7 and v.8 ROMs
0000                  ;ROM has simple programs to run on slow clock, with input switches and output LEDs
0000                  ;Also has monitor program with command to run CP/M
0000                  ;Jumps to address on input switches at power-on or reset
0000                  ;
0000                                    org   00000h
0000 db 04                              in    a,(4)            ;Get address from input ports
0002 6f                                 ld    l,a
0003 db 05                              in    a,(5)
0005 67                                 ld    h,a
0006 e9                                 jp    (hl)             ;Jump to the address
0007 db 04            Port_Reflector:   in    a,(4)            ;Simple program to test ports
0009 d3 04                              out   (4),a
000b db 05                              in    a,(5)
000d d3 05                              out   (5),a
000f c3 07 00                           jp    Port_Reflector
0012 3e 00            Simple_Counter:   ld    a,000h           ;One-byte counter for slow clock
0014 d3 04            Loop_1:           out   (4),a
0016 3c                                 inc   a
0017 c3 14 00                           jp    Loop_1
001a 2e 00            Count_to_a_million:     ld    l,000h     ;Two-byte (16-bit) counter
001c 26 00                              ld    h,000h           ;Clear registers
001e 3e 10            Loop_2:           ld    a,010h           ;Count 16 times, then
0020 3d               Loop_3:           dec   a
0021 c2 20 00                           jp    nz,Loop_3
0024 23                                 inc   hl               ;increment the 16-bit number
0025 7d                                 ld    a,l
0026 d3 04                              out   (4),a            ;Output the 16-bit number
0028 7c                                 ld    a,h
0029 d3 05                              out   (5),a
002b c3 1e 00                           jp    Loop_2           ;Do it again
002e
002e                  ;
002e                  ;
002e                  ;Subroutines for the monitor use these RAM variables:
```

```
002e            current_location: equu   0xdb00           ;word variable in RAM
002e            line_count:       equ   0xdb02           ;byte variable in RAM
002e            byte_count:       equ   0xdb03           ;byte variable in RAM
002e            value_pointer:          equ   0xdb04      ;word variable in RAM
002e            current_value:          equ   0xdb06      ;word variable in RAM
002e            buffer:                 equ   0xdb08      ;buffer in RAM -- up to stack area
002e
002e            ;Need to have stack in upper RAM, but not in area of CP/M or RAM monitor.
002e            ROM_monitor_stack:      equ   0xdbff      ;upper TPA in RAM, below RAM monitor
002e
002e            ;Subroutine to initialize serial port UART
002e            ;Needs to be called only once after computer comes out of reset.
002e            ;If called while port is active will cause port to fail.
002e            ;16x = 9600 baud
002e 3e 4e      initialize_port:  ld    a,04eh           ;1 stop bit, no parity, 8-bit char, 16x baud
0030 d3 03                        out   (3),a            ;write to control port
0032 3e 37                        ld    a,037h           ;enable receive and transmit
0034 d3 03                        out   (3),a            ;write to control port
0036 c9                                 ret
0037            ;
0037            ;Puts a single char (byte value) on serial output
0037            ;Call with char to send in A register. Uses B register
0037 47         write_char:       ld b,a           ;store char
0038 db 03      write_char_loop:  in    a,(3)            ;check if OK to send
003a e6 01                        and   001h             ;check TxRDY bit
003c ca 38 00                     jp z,write_char_loop   ;loop if not set
003f 78                           ld    a,b              ;get char back
0040 d3 02                        out   (2),a            ;send to output
0042 c9                           ret                    ;returns with char in a
0043            ;
0043            ;Subroutine to write a zero-terminated string to serial output
0043            ;Pass address of string in HL register
0043            ;No error checking
0043 db 03      write_string:     in    a,(3)            ;read status
0045 e6 01                        and   001h              ;check TxRDY bit
0047 ca 43 00                     jp    z,write_string   ;loop if not set
004a 7e                           ld    a,(hl)           ;get char from string
004b a7                           and   a                ;check if 0
004c c8                           ret   z                ;yes, finished
004d d3 02                        out   (2),a            ;no, write char to output
```

```
004f 23                     inc   hl                 ;next char in string
0050 c3 43 00               jp    write_string       ;start over
0053                  ;
0053                  ;Binary loader. Receive a binary file, place in memory.
0053                  ;Address of load passed in HL, length of load (= file length) in BC
0053 db 03            bload:        in    a,(3)              ;get status
0055 e6 02                  and   002h               ;check RxRDY bit
0057 ca 53 00               jp    z,bload            ;not ready, loop
005a db 02                  in    a,(2)
005c 77                     ld    (hl),a
005d 23                     inc   hl
005e 0b                     dec   bc                 ;byte counter
005f 78                     ld    a,b                ;need to test BC this way because
0060 b1                     or    c                  ;dec rp instruction does not change flags
0061 c2 53 00               jp    nz,bload
0064 c9                     ret
0065                  ;
0065                  ;Binary dump to port. Send a stream of binary data from memory to serial output
0065                  ;Address of dump passed in HL, length of dump in BC
0065 db 03            bdump:        in    a,(3)              ;get status
0067 e6 01                  and   001h               ;check TxRDY bit
0069 ca 65 00               jp    z,bdump            ;not ready, loop
006c 7e                     ld    a,(hl)
006d d3 02                  out   (2),a
006f 23                     inc   hl
0070 0b                     dec   bc
0071 78                     ld    a,b                ;need to test this way because
0072 b1                     or    c                  ;dec rp instruction does not change flags
0073 c2 65 00               jp    nz,bdump
0076 c9                     ret
0077                  ;
0077                  ;Subroutine to get a string from serial input, place in buffer.
0077                  ;Buffer address passed in HL reg.
0077                  ;Uses A,BC,DE,HL registers (including calls to other subroutines).
0077                  ;Line entry ends by hitting return key. Return char not included in string (replaced by zero).
0077                  ;Backspace editing OK. No error checking.
0077                  ;
0077 0e 00            get_line:     ld    c,000h             ;line position
0079 7c                     ld    a,h                ;put original buffer address in de
007a 57                     ld    d,a                ;after this don't need to preserve hl
```

```
007b 7d                         ld    a,l               ;subroutines called don't use de
007c 5f                         ld    e,a
007d db 03       get_line_next_char:     in    a,(3)       ;get status
007f e6 02                      and   002h              ;check RxRDY bit
0081 ca 7d 00                   jp    z,get_line_next_char    ;not ready, loop
0084 db 02                      in    a,(2)             ;get char
0086 fe 0d                      cp    00dh              ;check if return
0088 c8                         ret   z                 ;yes, normal exit
0089 fe 7f                      cp    07fh              ;check if backspace (VT102 keys)
008b ca 9f 00                   jp    z,get_line_backspace    ;yes, jump to backspace routine
008e fe 08                      cp    008h              ;check if backspace (ANSI keys)
0090 ca 9f 00                   jp    z,get_line_backspace    ;yes, jump to backspace
0093 cd 37 00                   call  write_char        ;put char on screen
0096 12                         ld    (de),a            ;store char in buffer
0097 13                         inc   de                ;point to next space in buffer
0098 0c                         inc   c                 ;inc counter
0099 3e 00                      ld    a,000h
009b 12                         ld    (de),a            ;leaves a zero-terminated string in buffer
009c c3 7d 00                   jp    get_line_next_char
009f 79          get_line_backspace:     ld    a,c         ;check current position in line
00a0 fe 00                      cp    000h              ;at beginning of line?
00a2 ca 7d 00                   jp    z,get_line_next_char    ;yes, ignore backspace, get next char
00a5 1b                         dec   de                ;no, erase char from buffer
00a6 0d                         dec   c                 ;back up one
00a7 3e 00                      ld    a,000h            ;put a zero in buffer where the last char was
00a9 12                         ld    (de),a
00aa 21 b5 03                   ld    hl,erase_char_string    ;ANSI sequence to delete one char from line
00ad cd 43 00                   call  write_string            ;transmits sequence to backspace and erase char
00b0 c3 7d 00                   jp    get_line_next_char
00b3             ;
00b3             ;Creates a two-char hex string from the byte value passed in register A
00b3             ;Location to place string passed in HL
00b3             ;String is zero-terminated, stored in 3 locations starting at HL
00b3             ;Also uses registers b,d, and e
00b3 47          byte_to_hex_string:     ld    b,a   ;store original byte
00b4 cb 3f                      srl   a                 ;shift right 4 times, putting
00b6 cb 3f                      srl   a                 ;high nybble in low-nybble spot
00b8 cb 3f                      srl   a                 ;and zeros in high-nybble spot
00ba cb 3f                      srl   a
00bc 16 00                      ld    d,000h            ;prepare for 16-bit addition
```

```
00be 5f                         ld    e,a               ;de contains offset
00bf e5                         push  hl                ;temporarily store string target address
00c0 21 19 01                   ld    hl,hex_char_table ;use char table to get high-nybble character
00c3 19                         add   hl,de             ;add offset to start of table
00c4 7e                         ld    a,(hl)            ;get char
00c5 e1                         pop   hl                ;get string target address
00c6 77                         ld    (hl),a            ;store first char of string
00c7 23                         inc   hl                ;point to next string target address
00c8 78                         ld    a,b               ;get original byte back from reg b
00c9 e6 0f                      and   00fh              ;mask off high-nybble
00cb 5f                         ld    e,a               ;d still has 000h, now de has offset
00cc e5                         push  hl                ;temp store string target address
00cd 21 19 01                   ld    hl,hex_char_table ;start of table
00d0 19                         add   hl,de             ;add offset
00d1 7e                         ld    a,(hl)            ;get char
00d2 e1                         pop   hl                ;get string target address
00d3 77                         ld    (hl),a            ;store second char of string
00d4 23                         inc   hl                ;point to third location
00d5 3e 00                      ld    a,000h            ;zero to terminate string
00d7 77                         ld    (hl),a            ;store the zero
00d8 c9                         ret                     ;done
00d9                  ;
00d9                  ;Converts a single ASCII hex char to a nybble value
00d9                  ;Pass char in reg A. Letter numerals must be upper case.
00d9                  ;Return nybble value in low-order reg A with zeros in high-order nybble if no error.
00d9                  ;Return 0ffh in reg A if error (char not a valid hex numeral).
00d9                  ;Also uses b, c, and hl registers.
00d9 21 19 01         hex_char_to_nybble:     ld    hl,hex_char_table
00dc 06 0f                      ld    b,00fh                  ;no. of valid characters in table - 1.
00de 0e 00                      ld    c,000h                  ;will be nybble value
00e0 be               hex_to_nybble_loop:     cp    (hl)            ;character match here?
00e1 ca ed 00                   jp    z,hex_to_nybble_ok      ;match found, exit
00e4 05                         dec   b                 ;no match, check if at end of table
00e5 fa ef 00                   jp    m,hex_to_nybble_err     ;table limit exceded, exit with error
00e8 0c                         inc   c                 ;still inside table, continue search
00e9 23                         inc   hl
00ea c3 e0 00                   jp    hex_to_nybble_loop
00ed 79               hex_to_nybble_ok: ld    a,c               ;put nybble value in a
00ee c9                         ret
00ef 3e ff            hex_to_nybble_err:      ld    a,0ffh            ;error value
```

```
00f1 c9                         ret
00f2             ;
00f2             ;Converts a hex character pair to a byte value
00f2             ;Called with location of high-order char in HL
00f2             ;If no error carry flag clear, returns with byte value in register A, and
00f2             ;HL pointing to next mem location after char pair.
00f2             ;If error (non-hex char) carry flag set, HL pointing to invalid char
00f2 7e          hex_to_byte:     ld    a,(hl)            ;location of character pair
00f3 e5                           push  hl                ;store hl (hex_char_to_nybble uses it)
00f4 cd d9 00                     call  hex_char_to_nybble
00f7 e1                           pop   hl                ;returns with nybble value in a reg, or 0ffh if error
00f8 fe ff                        cp    0ffh              ;non-hex character?
00fa ca 17 01                     jp    z,hex_to_byte_err ;yes, exit with error
00fd cb 27                        sla   a                 ;no, move low order nybble to high side
00ff cb 27                        sla   a
0101 cb 27                        sla   a
0103 cb 27                        sla   a
0105 57                           ld    d,a               ;store high-nybble
0106 23                           inc   hl                ;get next character of the pair
0107 7e                           ld    a,(hl)
0108 e5                           push  hl                ;store hl
0109 cd d9 00                     call  hex_char_to_nybble
010c e1                           pop   hl
010d fe ff                        cp    0ffh              ;non-hex character?
010f ca 17 01                     jp    z,hex_to_byte_err ;yes, exit with error
0112 b2                           or    d                 ;no, combine with high-nybble
0113 23                           inc   hl                ;point to next memory location after char pair
0114 37                           scf
0115 3f                           ccf                     ;no-error exit (carry = 0)
0116 c9                           ret
0117 37          hex_to_byte_err: scf                     ;error, carry flag set
0118 c9                           ret
0119 ..          hex_char_table:  defm  "0123456789ABCDEF"      ;ASCII hex table
0129             ;
0129             ;Subroutine to get a two-byte address from serial input.
0129             ;Returns with address value in HL
0129             ;Uses locations in RAM for buffer and variables
0129 21 08 db    address_entry:   ld    hl,buffer         ;location for entered string
012c cd 77 00                     call  get_line          ;returns with address string in buffer
012f 21 08 db                     ld    hl,buffer         ;location of stored address entry string
```

```
0132 cd f2 00                    call  hex_to_byte       ;will get high-order byte first
0135 da 4b 01                    jp    c, address_entry_error  ;if error, jump
0138 32 01 db                    ld    (current_location+1),a  ;store high-order byte, little-endian
013b 21 0a db                    ld    hl,buffer+2       ;point to low-order hex char pair
013e cd f2 00                    call  hex_to_byte       ;get low-order byte
0141 da 4b 01                    jp    c, address_entry_error  ;jump if error
0144 32 00 db                    ld    (current_location),a    ;store low-order byte in lower memory
0147 2a 00 db                    ld    hl,(current_location)   ;put memory address in hl
014a c9                          ret
014b 21 f3 03      address_entry_error:    ld    hl,address_error_msg
014e cd 43 00                    call  write_string
0151 c3 29 01                    jp    address_entry
0154               ;
0154               ;Subroutine to get a decimal string, return a word value
0154               ;Calls decimal_string_to_word subroutine
0154 21 08 db      decimal_entry:          ld    hl,buffer
0157 cd 77 00                          call  get_line          ;returns with DE pointing to terminating zero
015a 21 08 db                          ld    hl,buffer
015d cd 6a 01                          call  decimal_string_to_word
0160 d0                                ret   nc                ;no error, return with word in hl
0161 21 67 04                          ld    hl,decimal_error_msg    ;error, try again
0164 cd 43 00                          call  write_string
0167 c3 54 01                          jp    decimal_entry
016a               ;
016a               ;Subroutine to convert a decimal string to a word value
016a               ;Call with address of string in HL, pointer to end of string in DE
016a               ;Carry flag set if error (non-decimal char)
016a               ;Carry flag clear, word value in HL if no error.
016a 42            decimal_string_to_word: ld    b,d
016b 4b                                ld    c,e               ;use BC as string pointer
016c 22 00 db                          ld    (current_location),hl   ;store addr. of start of buffer in RAM
word variable
016f 21 00 00                          ld    hl,000h                 ;starting value zero
0172 22 06 db                          ld    (current_value),hl
0175 21 ba 01                          ld    hl,decimal_place_value  ;pointer to values
0178 22 04 db                          ld    (value_pointer),hl
017b 0b            decimal_next_char:      dec   bc                ;next char in string (moving right to left)
017c 2a 00 db                          ld    hl,(current_location)   ;check if at end of decimal string
017f 37                                scf                     ;get ready to subtract de from buffer addr.
0180 3f                                ccf                     ;set carry to zero (clear)
```

```
0181 ed 42                              sbc   hl,bc             ;keep going if bc > or = hl (buffer address)
0183 da 8f 01                           jp    c,decimal_continue      ;borrow means bc > hl
0186 ca 8f 01                           jp    z,decimal_continue      ;z means bc = hl
0189 2a 06 db                           ld    hl,(current_value)      ;return if de < buffer address (no borrow)
018c 37                                 scf                     ;get value back from RAM variable
018d 3f                                 ccf
018e c9                                 ret                     ;return with carry clear, value in hl
018f 0a           decimal_continue:     ld    a,(bc)            ;next char in string (right to left)
0190 d6 30                              sub   030h              ;ASCII value of zero char
0192 fa b5 01                           jp    m,decimal_error   ;error if char value less than 030h
0195 fe 0a                              cp    00ah              ;error if byte value > or = 10 decimal
0197 f2 b5 01                           jp    p,decimal_error   ;a reg now has value of decimal numeral
019a 2a 04 db                           ld    hl,(value_pointer)      ;get value to add an put in de
019d 5e                                 ld    e,(hl)            ;little-endian (low byte in low memory)
019e 23                                 inc   hl
019f 56                                 ld    d,(hl)
01a0 23                                 inc   hl                      ;hl now points to next value
01a1 22 04 db                           ld    (value_pointer),hl
01a4 2a 06 db                           ld    hl,(current_value)      ;get back current value
01a7 3d           decimal_add:          dec   a                       ;add loop to increase total value
01a8 fa af 01                           jp    m,decimal_add_done      ;end of multiplication
01ab 19                                 add   hl,de
01ac c3 a7 01                           jp    decimal_add
01af 22 06 db     decimal_add_done:     ld    (current_value),hl
01b2 c3 7b 01                           jp    decimal_next_char
01b5 37           decimal_error:        scf
01b6 c9                                 ret
01b7 c3 a7 01                           jp    decimal_add
01ba 01 00 0a 00 64 00 e8 03 10 27  decimal_place_value:   defw  1,10,100,1000,10000
01c4              ;
01c4              ;Memory dump
01c4              ;Displays a 256-byte block of memory in 16-byte rows.
01c4              ;Called with address of start of block in HL
01c4 22 00 db     memory_dump:          ld    (current_location),hl   ;store address of block to be displayed
01c7 3e 00                              ld    a,000h
01c9 32 03 db                           ld    (byte_count),a          ;initialize byte count
01cc 32 02 db                           ld    (line_count),a          ;initialize line count
01cf c3 04 02                           jp    dump_new_line
01d2 2a 00 db     dump_next_byte:       ld    hl,(current_location)   ;get byte address from storage,
01d5 7e                                 ld    a,(hl)                  ;get byte to be converted to string
```

```
01d6 23                                 inc   hl                      ;increment address and
01d7 22 00 db                           ld    (current_location),hl   ;store back
01da 21 08 db                           ld    hl,buffer               ;location to store string
01dd cd b3 00                           call  byte_to_hex_string      ;convert
01e0 21 08 db                           ld    hl,buffer               ;display string
01e3 cd 43 00                           call  write_string
01e6 3a 03 db                           ld    a,(byte_count)          ;next byte
01e9 3c                                 inc   a
01ea ca 34 02                           jp    z,dump_done             ;stop when 256 bytes displayed
01ed 32 03 db                           ld    (byte_count),a          ;not finished yet, store
01f0 3a 02 db                           ld    a,(line_count)          ;end of line (16 characters)?
01f3 fe 0f                              cp    00fh                    ;yes, start new line
01f5 ca 04 02                           jp    z,dump_new_line
01f8 3c                                 inc   a                       ;no, increment line count
01f9 32 02 db                           ld    (line_count),a
01fc 3e 20                              ld    a,020h                  ;print space
01fe cd 37 00                           call  write_char
0201 c3 d2 01                           jp    dump_next_byte          ;continue
0204 3e 00        dump_new_line:        ld    a,000h                  ;reset line count to zero
0206 32 02 db                           ld    (line_count),a
0209 cd ba 02                           call  write_newline
020c 2a 00 db                           ld    hl,(current_location)   ;location of start of line
020f 7c                                 ld    a,h               ;high byte of address
0210 21 08 db                           ld    hl, buffer
0213 cd b3 00                           call  byte_to_hex_string      ;convert
0216 21 08 db                           ld    hl,buffer
0219 cd 43 00                           call  write_string            ;write high byte
021c 2a 00 db                           ld    hl,(current_location)
021f 7d                                 ld    a,l               ;low byte of address
0220 21 08 db                           ld    hl, buffer
0223 cd b3 00                           call  byte_to_hex_string      ;convert
0226 21 08 db                           ld    hl,buffer
0229 cd 43 00                           call  write_string            ;write low byte
022c 3e 20                              ld    a,020h                  ;space
022e cd 37 00                           call  write_char
0231 c3 d2 01                           jp    dump_next_byte          ;now write 16 bytes
0234 3e 00        dump_done:            ld    a,000h
0236 21 08 db                           ld    hl,buffer
0239 77                                 ld    (hl),a                  ;clear buffer of last string
023a cd ba 02                           call  write_newline
```

```
023d c9                                 ret
023e             ;
023e             ;Memory load
023e             ;Loads RAM memory with bytes entered as hex characters
023e             ;Called with address to start loading in HL
023e             ;Displays entered data in 16-byte rows.
023e 22 00 db    memory_load:           ld    (current_location),hl
0241 21 1f 04                           ld    hl,data_entry_msg
0244 cd 43 00                           call  write_string
0247 c3 97 02                           jp    load_new_line
024a cd b0 02    load_next_char:        call  get_char
024d fe 0d                              cp    00dh                ;return?
024f ca ac 02                           jp    z,load_done         ;yes, quit
0252 32 08 db                           ld    (buffer),a
0255 cd b0 02                           call  get_char
0258 fe 0d                              cp    00dh                ;return?
025a ca ac 02                           jp    z,load_done         ;yes, quit
025d 32 09 db                           ld    (buffer+1),a
0260 21 08 db                           ld    hl,buffer
0263 cd f2 00                           call  hex_to_byte
0266 da a2 02                           jp    c,load_data_entry_error ;non-hex character
0269 2a 00 db                           ld    hl,(current_location)   ;get byte address from storage,
026c 77                                 ld    (hl),a                  ;store byte
026d 23                                 inc   hl                  ;increment address and
026e 22 00 db                           ld    (current_location),hl   ;store back
0271 3a 08 db                           ld    a,(buffer)
0274 cd 37 00                           call  write_char
0277 3a 09 db                           ld    a,(buffer+1)
027a cd 37 00                           call  write_char
027d 3a 02 db                           ld    a,(line_count)      ;end of line (16 characters)?
0280 fe 0f                              cp    00fh                ;yes, start new line
0282 ca 97 02                           jp    z,load_new_line
0285 3c                                 inc   a                   ;no, increment line count
0286 32 02 db                           ld    (line_count),a
0289 3e 20                              ld    a,020h              ;print space
028b cd 37 00                           call  write_char
028e c3 4a 02                           jp    load_next_char      ;continue
0291             ;This section is to align the jump to disk_read for the cpm_loader
0291             ;Expects disk_read to be at location 0x0294
0291 00                                 nop
```

```
0292 00                              nop
0293 00                              nop
0294 c3 c5 02                        jp    disk_read
0297 3e 00       load_new_line:      ld    a,000h                  ;reset line count to zero
0299 32 02 db                        ld    (line_count),a
029c cd ba 02                        call  write_newline
029f c3 4a 02                        jp    load_next_char          ;continue
02a2 cd ba 02    load_data_entry_error:  call  write_newline
02a5 21 4c 04                        ld    hl,data_error_msg
02a8 cd 43 00                        call  write_string
02ab c9                              ret
02ac cd ba 02    load_done:          call  write_newline
02af c9                              ret
02b0             ;
02b0             ;Get one ASCII character from the serial port.
02b0             ;Returns with char in A reg. No error checking.
02b0 db 03       get_char:      in    a,(3)             ;get status
02b2 e6 02                      and   002h              ;check RxRDY bit
02b4 ca b0 02                   jp    z,get_char        ;not ready, loop
02b7 db 02                      in    a,(2)             ;get char
02b9 c9                              ret
02ba             ;
02ba             ;Subroutine to start a new line
02ba 3e 0d       write_newline:      ld    a,00dh            ;ASCII carriage return character
02bc cd 37 00                        call  write_char
02bf 3e 0a                           ld    a,00ah            ;new line (line feed) character
02c1 cd 37 00                        call  write_char
02c4 c9                              ret
02c5             ;
02c5             ;Subroutine to read one disk sector (256 bytes)
02c5             ;Address to place data passed in HL
02c5             ;LBA bits 0 to 7 passed in C, bits 8 to 15 passed in B
02c5             ;LBA bits 16 to 23 passed in E
02c5             disk_read:
02c5 db 0f       rd_status_loop_1: in    a,(0fh)           ;check status
02c7 e6 80                      and   80h               ;check BSY bit
02c9 c2 c5 02                   jp    nz,rd_status_loop_1     ;loop until not busy
02cc db 0f       rd_status_loop_2: in    a,(0fh)           ;check      status
02ce e6 40                      and   40h               ;check DRDY bit
02d0 ca cc 02                   jp    z,rd_status_loop_2      ;loop until ready
```

```
02d3 3e 01                      ld    a,01h            ;number of sectors = 1
02d5 d3 0a                      out   (0ah),a          ;sector count register
02d7 79                         ld    a,c
02d8 d3 0b                      out   (0bh),a          ;lba bits 0 - 7
02da 78                         ld    a,b
02db d3 0c                      out   (0ch),a          ;lba bits 8 - 15
02dd 7b                         ld    a,e
02de d3 0d                      out   (0dh),a          ;lba bits 16 - 23
02e0 3e e0                      ld    a,11100000b      ;LBA mode, select drive 0
02e2 d3 0e                      out   (0eh),a          ;drive/head register
02e4 3e 20                      ld    a,20h            ;Read sector command
02e6 d3 0f                      out   (0fh),a
02e8 db 0f       rd_wait_for_DRQ_set:    in    a,(0fh)     ;read status
02ea e6 08                      and   08h              ;DRQ bit
02ec ca e8 02                   jp    z,rd_wait_for_DRQ_set   ;loop until bit set
02ef db 0f       rd_wait_for_BSY_clear:  in    a,(0fh)
02f1 e6 80                      and   80h
02f3 c2 ef 02                   jp    nz,rd_wait_for_BSY_clear
02f6 db 0f                      in    a,(0fh)          ;clear INTRQ
02f8 db 08       read_loop:     in    a,(08h)          ;get data
02fa 77                         ld    (hl),a
02fb 23                         inc   hl
02fc db 0f                      in    a,(0fh)          ;check status
02fe e6 08                      and   08h              ;DRQ bit
0300 c2 f8 02                   jp    nz,read_loop     ;loop until cleared
0303 c9                         ret
0304             ;
0304             ;Subroutine to write one disk sector (256 bytes)
0304             ;Address of data to write to disk passed in HL
0304             ;LBA bits 0 to 7 passed in C, bits 8 to 15 passed in B
0304             ;LBA bits 16 to 23 passed in E
0304             disk_write:
0304 db 0f       wr_status_loop_1: in    a,(0fh)       ;check status
0306 e6 80                      and   80h              ;check BSY bit
0308 c2 04 03                   jp    nz,wr_status_loop_1     ;loop until not busy
030b db 0f       wr_status_loop_2: in    a,(0fh)       ;check status
030d e6 40                      and   40h              ;check DRDY bit
030f ca 0b 03                   jp    z,wr_status_loop_2      ;loop until ready
0312 3e 01                      ld    a,01h            ;number of sectors = 1
0314 d3 0a                      out   (0ah),a          ;sector count register
```

```
0316 79                          ld    a,c
0317 d3 0b                       out   (0bh),a           ;lba bits 0 - 7
0319 78                          ld    a,b
031a d3 0c                       out   (0ch),a           ;lba bits 8 - 15
031c 7b                          ld    a,e
031d d3 0d                       out   (0dh),a           ;lba bits 16 - 23
031f 3e e0                       ld    a,11100000b       ;LBA mode, select drive 0
0321 d3 0e                       out   (0eh),a           ;drive/head register
0323 3e 30                       ld    a,30h             ;Write sector command
0325 d3 0f                       out   (0fh),a
0327 db 0f       wr_wait_for_DRQ_set:    in    a,(0fh)     ;read status
0329 e6 08                       and   08h               ;DRQ bit
032b ca 27 03                    jp    z,wr_wait_for_DRQ_set   ;loop until bit set
032e 7e          write_loop:     ld    a,(hl)
032f d3 08                       out   (08h),a           ;write data
0331 23                          inc   hl
0332 db 0f                       in    a,(0fh)           ;read status
0334 e6 08                       and   08h               ;check DRQ bit
0336 c2 2e 03                    jp    nz,write_loop     ;write until bit cleared
0339 db 0f       wr_wait_for_BSY_clear:  in    a,(0fh)
033b e6 80                       and   80h
033d c2 39 03                    jp    nz,wr_wait_for_BSY_clear
0340 db 0f                       in    a,(0fh)           ;clear INTRQ
0342 c9                          ret
0343             ;
0343             ;Strings used in subroutines
0343 .. 00       length_entry_string:    defm  "Enter length of file to load (decimal): ",0
036c .. 00       dump_entry_string:      defm  "Enter no. of bytes to dump (decimal): ",0
0393 .. 00       LBA_entry_string: defm  "Enter LBA (decimal, 0 to 65535): ",0
03b5 08 1b .. 00  erase_char_string:     defm  008h,01bh,"[K",000h     ;ANSI sequence for backspace, erase to end
of line.
03ba .. 00       address_entry_msg:      defm  "Enter 4-digit hex address (use upper-case A through F): ",0
03f3 .. 00       address_error_msg:      defm  "\r\nError: invalid hex character, try again: ",0
041f .. 00       data_entry_msg:         defm  "Enter hex bytes, hit return when finished.\r\n",0
044c .. 00       data_error_msg:         defm  "Error: invalid hex byte.\r\n",0
0467 .. 00       decimal_error_msg:      defm  "\r\nError: invalid decimal number, try again: ",0
0494             ;
0494             ;Simple monitor program for CPUville Z80 computer with serial interface.
0494 31 ff db    monitor_cold_start:     ld    sp,ROM_monitor_stack
0497 cd 2e 00                           call  initialize_port
```

```
049a 21 12 06                           ld    hl,monitor_message
049d cd 43 00                           call  write_string
04a0 cd ba 02     monitor_warm_start:   call  write_newline      ;routine program return here to avoid re-
initialization of port
04a3 3e 3e                              ld    a,03eh             ;prompt symbol
04a5 cd 37 00                           call  write_char
04a8 21 08 db                           ld    hl,buffer
04ab cd 77 00                           call  get_line           ;get monitor input string (command)
04ae cd ba 02                           call  write_newline
04b1 cd b5 04                           call  parse              ;interprets command, returns with address to
jump to in HL
04b4 e9                                 jp    (hl)
04b5              ;
04b5              ;Parses an input line stored in buffer for available commands as described in parse table.
04b5              ;Returns with address of jump to action for the command in HL
04b5 01 be 07     parse:                ld    bc,parse_table     ;bc is pointer to parse_table
04b8 0a           parse_start:          ld    a,(bc)             ;get pointer to match string from parse table
04b9 5f                                 ld    e,a
04ba 03                                 inc   bc
04bb 0a                                 ld    a,(bc)
04bc 57                                 ld    d,a                ;de will is pointer to strings for matching
04bd 1a                                 ld    a,(de)             ;get first char from match string
04be f6 00                              or    000h               ;zero?
04c0 ca db 04                           jp    z,parser_exit      ;yes, exit no_match
04c3 21 08 db                           ld    hl,buffer          ;no, parse input string
04c6 be           match_loop:           cp    (hl)               ;compare buffer char with match string char
04c7 c2 d5 04                           jp    nz,no_match        ;no match, go to next match string
04ca f6 00                              or    000h               ;end of strings (zero)?
04cc ca db 04                           jp    z,parser_exit      ;yes, matching string found
04cf 13                                 inc   de                 ;match so far, point to next char in match
string
04d0 1a                                 ld    a,(de)             ;get next character from match string
04d1 23                                 inc   hl                 ;and point to next char in input string
04d2 c3 c6 04                           jp    match_loop         ;check for match
04d5 03           no_match:             inc   bc                 ;skip over jump target to
04d6 03                                 inc   bc
04d7 03                                 inc   bc                 ;get address of next matching string
04d8 c3 b8 04                           jp    parse_start
04db 03           parser_exit:          inc   bc                 ;skip to address of jump for match
04dc 0a                                 ld    a,(bc)
```

```
04dd 6f                         ld    l,a
04de 03                         inc   bc
04df 0a                         ld    a,(bc)
04e0 67                         ld    h,a              ;returns with jump address in hl
04e1 c9                         ret
04e2               ;
04e2               ;Actions to be taken on match
04e2               ;
04e2               ;Memory dump program
04e2               ;Input 4-digit hexadecimal address
04e2               ;Calls memory_dump subroutine
04e2 21 31 06      dump_jump:       ld    hl,dump_message        ;Display greeting
04e5 cd 43 00                   call  write_string
04e8 21 ba 03                   ld    hl,address_entry_msg   ;get ready to get address
04eb cd 43 00                   call  write_string
04ee cd 29 01                   call  address_entry          ;returns with address in HL
04f1 cd ba 02                   call  write_newline
04f4 cd c4 01                   call  memory_dump
04f7 c3 a0 04                   jp    monitor_warm_start
04fa               ;
04fa               ;Hex loader, displays formatted input
04fa 21 51 06      load_jump:       ld    hl,load_message        ;Display greeting
04fd cd 43 00                   call  write_string           ;get address to load
0500 21 ba 03                   ld    hl,address_entry_msg   ;get ready to get address
0503 cd 43 00                   call  write_string
0506 cd 29 01                   call  address_entry
0509 cd ba 02                   call  write_newline
050c cd 3e 02                   call  memory_load
050f c3 a0 04                   jp    monitor_warm_start
0512               ;
0512               ;Jump and run do the same thing: get an address and jump to it.
0512 21 6e 06      run_jump:        ld    hl,run_message         ;Display greeting
0515 cd 43 00                   call  write_string
0518 21 ba 03                   ld    hl,address_entry_msg   ;get ready to get address
051b cd 43 00                   call  write_string
051e cd 29 01                   call  address_entry
0521 e9                         jp    (hl)
0522               ;
0522               ;Help and ? do the same thing, display the available commands
0522 21 25 06      help_jump:       ld    hl,help_message
```

```
0525 cd 43 00                      call  write_string
0528 01 be 07                      ld    bc,parse_table    ;table with pointers to command strings
052b 0a          help_loop:        ld    a,(bc)            ;displays the strings for matching commands,
052c 6f                            ld    l,a               ;getting the string addresses from the
052d 03                            inc   bc                ;parse table
052e 0a                            ld    a,(bc)            ;pass address of string to hl through a reg
052f 67                            ld    h,a
0530 7e                            ld    a,(hl)            ;hl now points to start of match string
0531 f6 00                         or    000h              ;exit if no_match string
0533 ca 46 05                      jp    z,help_done
0536 c5                            push  bc                ;write_char uses b register
0537 3e 20                         ld    a,020h            ;space char
0539 cd 37 00                      call  write_char
053c c1                            pop   bc
053d cd 43 00                      call  write_string      ;writes match string
0540 03                            inc   bc                ;pass over jump address in table
0541 03                            inc   bc
0542 03                            inc   bc
0543 c3 2b 05                      jp    help_loop
0546 c3 a0 04    help_done:        jp    monitor_warm_start
0549             ;
0549             ;Binary file load. Need both address to load and length of file
0549 21 95 06    bload_jump:       ld    hl,bload_message
054c cd 43 00                      call  write_string
054f 21 ba 03                      ld    hl,address_entry_msg
0552 cd 43 00                      call  write_string
0555 cd 29 01                      call  address_entry
0558 cd ba 02                      call  write_newline
055b e5                            push  hl
055c 21 43 03                      ld    hl,length_entry_string
055f cd 43 00                      call  write_string
0562 cd 54 01                      call  decimal_entry
0565 44                            ld    b,h
0566 4d                            ld    c,l
0567 21 b8 06                      ld    hl,bload_ready_message
056a cd 43 00                      call  write_string
056d e1                            pop   hl
056e cd 53 00                      call  bload
0571 c3 a0 04                      jp    monitor_warm_start
0574             ;
```

```
0574                ;Binary memory dump. Need address of start of dump and no. bytes
0574 21 dc 06       bdump_jump:             ld    hl,bdump_message
0577 cd 43 00                               call  write_string
057a 21 ba 03                               ld    hl,address_entry_msg
057d cd 43 00                               call  write_string
0580 cd 29 01                               call  address_entry
0583 cd ba 02                               call  write_newline
0586 e5                                     push  hl
0587 21 6c 03                               ld    hl,dump_entry_string
058a cd 43 00                               call  write_string
058d cd 54 01                               call  decimal_entry
0590 44                                     ld    b,h
0591 4d                                     ld    c,l
0592 21 0c 07                               ld    hl,bdump_ready_message
0595 cd 43 00                               call  write_string
0598 cd b0 02                               call  get_char
059b e1                                     pop   hl
059c cd 65 00                               call  bdump
059f c3 a0 04                               jp    monitor_warm_start
05a2                ;Disk read. Need memory address to place data, LBA of sector to read
05a2 21 33 07       diskrd_jump:            ld    hl,diskrd_message
05a5 cd 43 00                               call  write_string
05a8 21 ba 03                               ld    hl,address_entry_msg
05ab cd 43 00                               call  write_string
05ae cd 29 01                               call  address_entry
05b1 cd ba 02                               call  write_newline
05b4 e5                                     push  hl
05b5 21 93 03                               ld    hl,LBA_entry_string
05b8 cd 43 00                               call  write_string
05bb cd 54 01                               call  decimal_entry
05be 44                                     ld    b,h
05bf 4d                                     ld    c,l
05c0 1e 00                                  ld    e,00h
05c2 e1                                     pop   hl
05c3 cd c5 02                               call  disk_read
05c6 c3 a0 04                               jp    monitor_warm_start
05c9 21 5b 07       diskwr_jump:            ld    hl,diskwr_message
05cc cd 43 00                               call  write_string
05cf 21 ba 03                               ld    hl,address_entry_msg
05d2 cd 43 00                               call  write_string
```

```
05d5 cd 29 01                              call  address_entry
05d8 cd ba 02                              call  write_newline
05db e5                                    push  hl
05dc 21 93 03                              ld    hl,LBA_entry_string
05df cd 43 00                              call  write_string
05e2 cd 54 01                              call  decimal_entry
05e5 44                                    ld    b,h
05e6 4d                                    ld    c,l
05e7 1e 00                                 ld    e,00h
05e9 e1                                    pop   hl
05ea cd 04 03                              call  disk_write
05ed c3 a0 04                              jp    monitor_warm_start
05f0 21 00 08           cpm_jump:          ld    hl,0800h
05f3 01 00 00                              ld    bc,0000h
05f6 1e 00                                 ld    e,00h
05f8 cd c5 02                              call  disk_read
05fb c3 00 08                              jp    0800h
05fe                ;Prints message for no match to entered command
05fe 21 1d 06           no_match_jump:     ld    hl,no_match_message
0601 cd 43 00                              call  write_string
0604 21 08 db                              ld    hl, buffer
0607 cd 43 00                              call  write_string
060a c3 a0 04                              jp    monitor_warm_start
060d                ;
060d                ;Monitor data structures:
060d                ;
060d .. 00          monitor_message:       defm  "\r\nROM ver. 15\r\n",0
061d .. 00          no_match_message:      defm  "? ",0
0620 .. 00          help_message:          defm  "Commands:\r\n",0
062c .. 00          dump_message:          defm  "Displays 256 bytes of memory.\r\n",0
064c .. 00          load_message:          defm  "Enter hex bytes in memory.\r\n",0
0669 .. 00          run_message:           defm  "Will run program at address entered.\r\n",0
0690 .. 00          bload_message:         defm  "Loads a binary file into memory.\r\n",0
06b3 .. 00          bload_ready_message:   defm  "\n\rReady to receive, start transfer.",0
06d7 .. 00          bdump_message:         defm  "Dumps binary data from memory to serial port.\r\n",0
0707 .. 00          bdump_ready_message:   defm  "\n\rReady to send, hit any key to start.",0
072e .. 00          diskrd_message:        defm  "Reads one sector from disk to memory.\r\n",0
0756 .. 00          diskwr_message:        defm  "Writes one sector from memory to disk.\r\n",0
077f                ;Strings for matching:
077f .. 00          dump_string:           defm  "dump",0
```

```
0784 .. 00        load_string:           defm  "load",0
0789 .. 00        jump_string:           defm  "jump",0
078e .. 00        run_string:            defm  "run",0
0792 .. 00        question_string:       defm  "?",0
0794 .. 00        help_string:           defm  "help",0
0799 .. 00        bload_string:          defm  "bload",0
079f .. 00        bdump_string:          defm  "bdump",0
07a5 .. 00        diskrd_string:         defm  "diskrd",0
07ac .. 00        diskwr_string:         defm  "diskwr",0
07b3 .. 00        cpm_string:            defm  "cpm",0
07b7 00 00        no_match_string:       defm  0,0
07b9              ;Table for matching strings to jumps
07b9 7f 07 e2 04 84 07 fa 04  parse_table:     defw  dump_string,dump_jump,load_string,load_jump
07c1 89 07 12 05 8e 07 12 05                   defw  jump_string,run_jump,run_string,run_jump
07c9 92 07 22 05 94 07 22 05                   defw  question_string,help_jump,help_string,help_jump
07d1 99 07 49 05 9f 07 74 05                   defw  bload_string,bload_jump,bdump_string,bdump_jump
07d9 a5 07 a2 05 ac 07 c9 05                   defw  diskrd_string,diskrd_jump,diskwr_string,diskwr_jump
07e1 b3 07 f0 05                               defw  cpm_string,cpm_jump
07e5 b7 07 fe 05                               defw  no_match_string,no_match_jump
07e9
# End of file 2K_ROM_15.asm
07e9
```