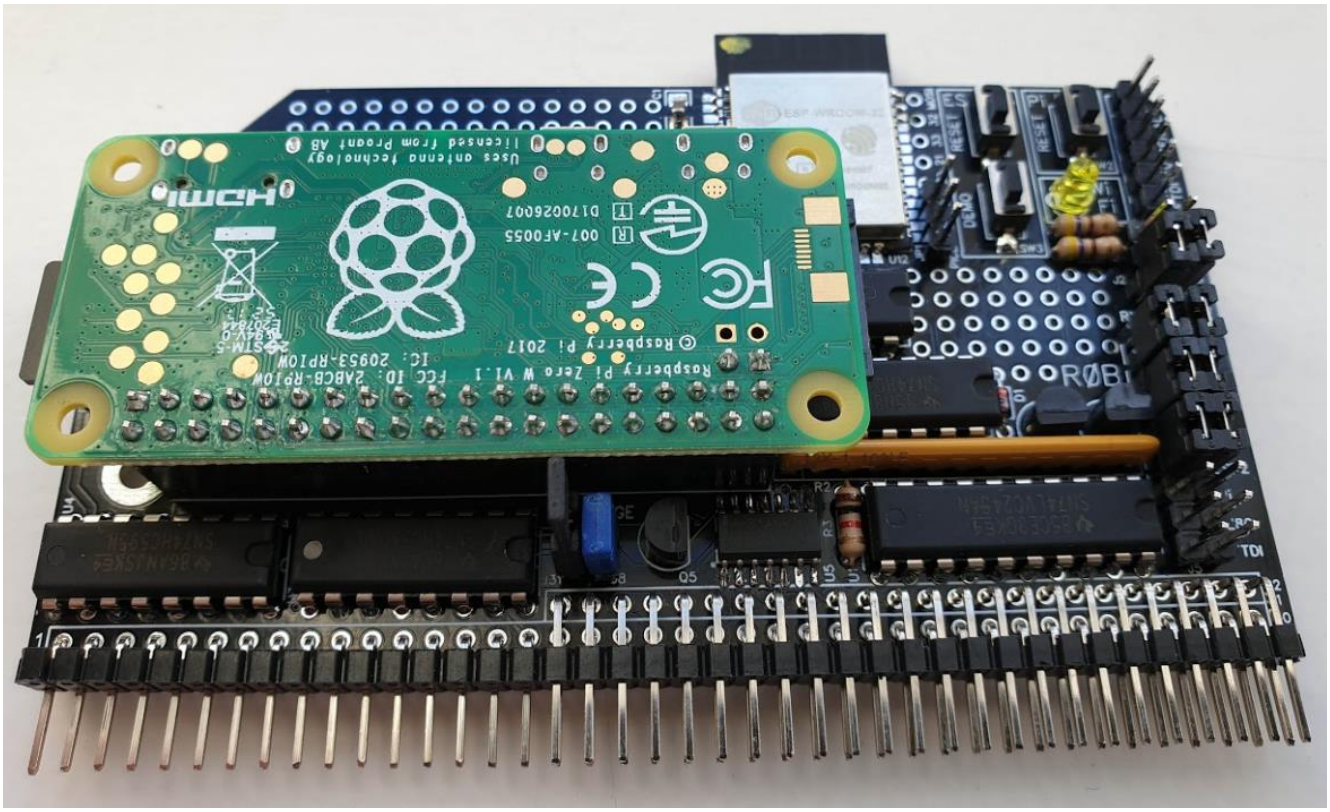
BUSRAIDER 1.7.133 MANUAL

INTRODUCTION TO THE BUSRAIDER

The BusRaider is a new way to run retro software, including games, and is neither a clone of a retro computer or a pure emulator but a blend of the two. It uses a real retro computer (the RC2014) as its base, so the code runs on a genuine Z80 but the aspects which are unique to the specific retro computer (such as memory-mapped display, keyboard, etc) are emulated by code running in bare-metal mode on a Raspberry Pi Zero. In addition, the BusRaider enables single-step debugging of Z80 programs.



BUSRAIDER FUNCTIONALITY

The BusRaider is designed to plug into the (PRO) bus of the RC2014 Retro-Computer (https://rc2014.co.uk/) and provide the following:

- Emulation of the memory-mapped video and keyboard functionality of some 1970's/80's era computers such as the Sinclair ZX Spectrum and Tandy TRS-80
- Single-step debugging of Z80 programs (including breakpoints, register and memory inspection)
- Tracing of every bus access the Z80 makes with streaming over WiFi
- WiFi with a web-based UI to allow programs to be quickly run and managed
- Serial terminal on Pi display with a mirror onto the Web UI
- Emulation of ANSI and H19 terminal escape codes including 8/16/256 colour support
- Telnet support for local or remote access to a serial-enabled machine (or 6850 emulator)
- Demo-mode to allow you to show off your RC2014's amazing abilities at the click of a button!
- Bus control functionality allowing REST API to be used to set addr, data, MREQ strobe, WR, etc
-

To achieve all of this functionality the BusRaider has two main tricks up its sleeve:

- Accessing the RC2014's bus using the BUSRQ line. Once requested and acknowledged the Z80 "lets go" of all the bus lines and allows the BusRaider to read/write memory and IO without interference.
- Detecting when the Z80 is performing a memory/IO operation or responding to an interrupt and then forcing the Z80 to wait so that the BusRaider can:
    - respond to the memory/IO operation itself (emulating IO or memory)
    - single-step through a program
    - detect debug breakpoints and pause execution at that point

THIS IS A HOBBY PROJECT

While the hardware of the BusRaider is relatively simple the vast majority of effort in this project has been developing software for the Raspberry Pi Zero (W) to control the hardware in real-time to achieve the desired end-goals. While much effort has been made to ensure that this operates without error in all cases you should note that this is a hobby project and no guarantees can be made about the reliability and robustness of the hardware or software. If you do discover any problems, please do let me know. Ideally this should be via GitHub's issue tracking mechanism at https://github.com/robdobsn/PiBusRaider/issues

PREREQUISITES

In order for the BusRaider to operate (in any mode) a few things need to be ensured:

- The BusRaider is in a PRO backplane (such as Spencer's https://rc2014.co.uk/modules/backplane-pro/ or Steve's https://www.tindie.com/products/tindiescx/modular-backplane-boards-for-rc2014/) as additional bus lines BUSRQ, BUSACK, WAIT, NMI, RST2(Page) are used
- Jumpers need to be in place to enable communication between the ESP32 and Pi (at a minimum). The position of these jumpers is shown in Appendix 8.2
- A Z80 processor board suitable for use with the PRO backplane – e.g. Spencer's Z80 CPU 2.1 https://rc2014.co.uk/modules/cpu/z80-cpu-v2-1/
- RAM and/or ROM cards for the RC2014 bus. Suitable cards (see options below) include Spencer's 512K RAM/ROM card and his 64K RAM card.
- USB keyboard and adapter for micro USB connector on the Pi Zero (W) – to interact with the target computer software and also set the WiFi settings
- HDMI monitor and adaptor for the small HDMI connector on the Pi Zero (W) – to see the main output from the BusRaider and memory-mapped graphics

Optionally (depending on how you plan to use the BusRaider) you should also have the following:

- Modifications to RAM/ROM cards if required to support paging. If you wish to use opcode injection or the debugging facilities of the BusRaider then:
    - the RAM / ROM cards you choose should have paging support (see Appendix 13 for instructions on modifying the 64K RAM card)
    - the PAGE jumper should be fitted on the BusRaider – see Appendix 13.
- Z80 clock module. This is optional (and NOT recommended) because the BusRaider can generate an appropriate clock for the Z80 and this can be varied to approximate the speed of an original computer such as ZX Spectrum or TRS80. If a clock module is used then the CLK jumper on the BusRaider should NOT be fitted.

- Z80 serial port (68B50 / Z80 SIO / etc) card. This is optionally used if serial-based software is to be used with the BusRaider such as the BASIC or CP/M software provided in various forms with versions of the RC2014 ROM/CF-CARD/etc. If this is to be used the serial jumpers need to be installed to connect the Z80 serial port (1 or 2) to the ESP32 as described in Appendix 8. Note that an emulation of the 6850 is also available to avoid the need for serial cards.
- FTDI Serial cable (3V) – this can be used to set WiFi settings on the ESP32, reprogram the ESP32 and for diagnostic purposes as an initial check of board functionality or if there are operational issues
- Visual Studio add-on called Z80 Debug (https://github.com/maziac/z80-debug) created by Maziac which performs single-step debugging of Z80 programs

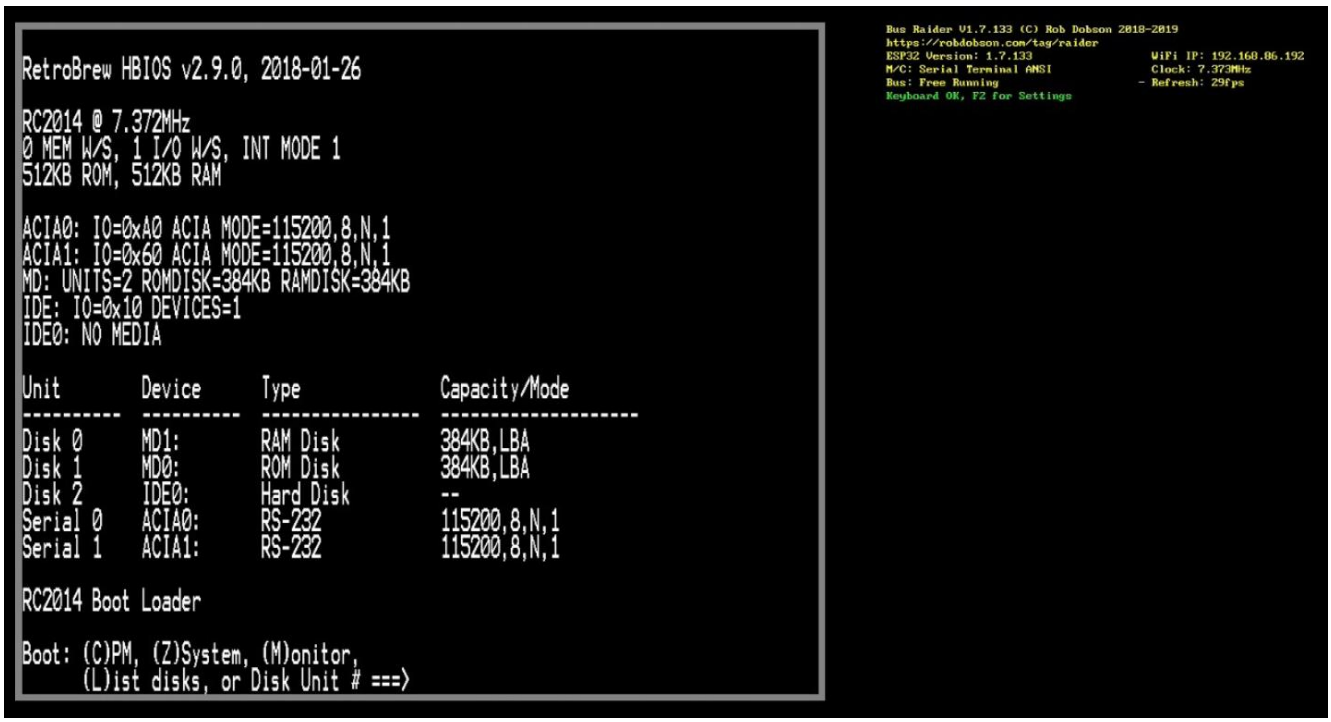
GETTING STARTED

NOTE: There is one component change: R3 should be 1K instead of 10K

If you haven't built or tested the BusRaider yet, then please refer to Appendix 10 Construction and/or Appendix 11 Inspection and Testing at this stage.

Continue with the Initial setup below once you have the BusRaider at the point where the HDMI output from the Raspberry Pi Zero (W) is showing the presence of the ESP32 – see below.

In addition, if you have a keyboard connected to the Raspberry Pi Zero (W) USB port, then you should see confirmation of this in the display as shown below (note that the part on the right is the area of interest – the left may look rather different depending on what machine is selected, whether it is configured and running, etc.



Things to note here are:

- A keyboard should be found if plugged into the Pi USB connector (not the one marked PWR IN)

- The ESP32 version should show up on the right-side status information. If you see “Not Connected!” beside ESP32 Version then check the ESP32 jumpers explained in Appendix 8.2

INITIAL SETUP

The first thing to do is to get the BusRaider onto your WiFi network. There are two ways to do this:

- Using the keyboard and HDMI display on the Raspberry Pi. First press F2 on the USB keyboard to enter “Immediate Mode”. You should see confirmation on the HDMI display. Then follow the instructions below.
- Using an FTDI serial connection cable as described in Appendix 9 with jumpers in place for FTDI to ESP32 as described in Appendix 8. Open terminal emulator software (Appendix 9) and make sure you can see the diagnostic messages described in the Appendix 12. Then follow the instructions below.

Enter (with either the USB keyboard or in the terminal emulator) your WiFi network SSID and password together with an optional “hostname” as follows:

w/YOURSSID/YOURPASSWORD/YOURHOSTNAME

Where:

- w starts the command to set up the wireless network
- / is used as a separator between the parameters to the command
- YOURSSID is the SSID of your network (doesn’t have to be all caps I’ve just done that for effect)
- YOURPASSWORD is the password to your network. At present this assumes the network is secure.
- YOURHOSTNAME is the hostname you want the BusRaider to appear as on your network. So that, for instance, if you chose the hostname busraider then when you are using the chrome browser (although maybe not other browsers) you can type just busraider into the address bar and see the Web UI for the BusRaider appear.

For example you may enter:

w/MyWifi/mypassword/busraider

After you have entered the SSID wait a short time and you should see the HDMI display update with the IP address that the BusRaider’s ESP32 has been assigned by your network. See the screen shot in the previous section under WiFi IP: on the right-side status panel. If this doesn’t happen then you could check the output on the terminal emulator connected via the FTDI cable (if you have one) or just try again in case you entered an incorrect ssid or password.

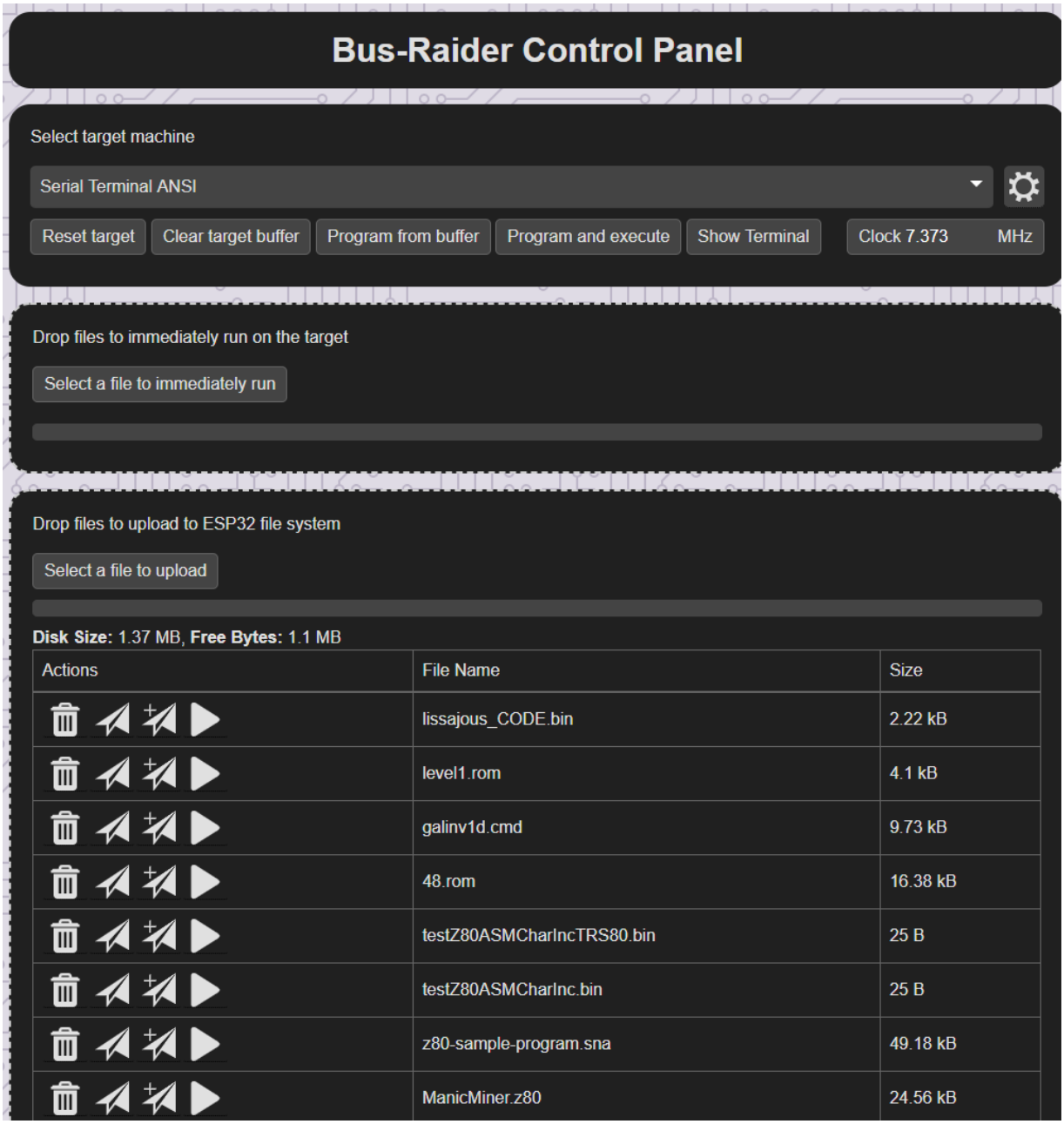
Note that the WiFi support doesn’t work with all network security schemes (Enterprise security isn’t supported for instance) and there may be issues with spaces or other odd characters in the password or SSID.

THE BUSRAIDER WEB UI

Now that the BusRaider is on the WiFi network we can look at its WebUI. Open a browser (preferably chrome as it is generally able to use the hostname we set up with the WiFi settings) and enter one of the following:

- The hostname you chose when assigning WiFi settings (maybe you followed the example I gave and entered busraider)
- The IP address that your network has assigned to the BusRaider – this will be shown on the HDMI display in the yellow status text.

In either case you should see the WebUI which should look something like this (although the file list at the bottom will probably be empty):



The main sections of this UI are as follows:

- Select target machine … target machine type with gear at the right for settings
- Drop files to immediately run … file drop area for immediate uploads
- Drop files to upload to ESP32 file system … file system contents and buttons to run programs, etc
- Show Terminal button to display Serial Terminal mirrored onto WebUI
- Clock speed setting

SELECTING THE TARGET MACHINE

The target machines currently supported are as follows:

- ZX Spectrum – a partial implementation of emulation for the ZX Spectrum. It provides display graphics (currently excluding border colour handling), keyboard and interrupt emulation (without accurate timing).

Some Spectrum games have been tested and seem to work ok – others not so much. File format support for SNA, Z80 and ZTX formats have been attempted with mixed success.

- TRS80 – seems to work pretty well with level1 ROM functionality and games such as Galaxy Invasion have been played extensively – purely for testing purposes of course 😊
- Serial Terminal – this machine emulates a serial terminal (ANSI and H19 escape support) on the HDMI output of the Pi (with mirroring to the WebUI). In addition Telnet is supported over WiFi to allow a session to be run from a remote machine. Also, a UART is not strictly necessary on the target machine as the 6850 is (optionally) emulated.

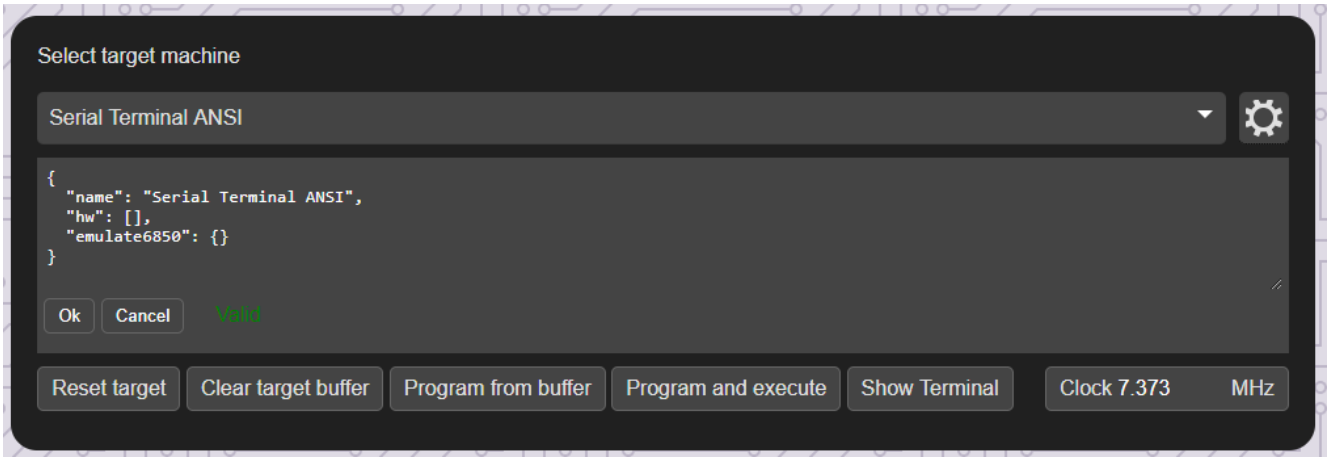
You will probably have a favourite machine that you would like to use at this stage so select this from the drop-down list. Also select the options you require from he checkboxes below the target machine. The options are described more fully in Appendix 4 and note that your hardware should match the options you select – specifically:

- A pure RAM board (e.g. 64K RAM with Paging – see Appendix 13) is a good choice of hardware if you are wanting to play retro games. In this case the ROM image and game image can both be loaded into RAM through the WebUI.
- A ROM/RAM board such as Spencer's 512K RAM/ROM is a good option if you want to use Serial Terminal support.
- if you want to use the single-step debugging capability then your RAM/ROM cards should support paging using the PAGE (RST2) line on the RC2014 PRO bus, the PAGE jumper should be fitted on the BusRaider PCB and there must be a pull-up resistor on the PAGE (RST2) line as described in Appendix 13

SELECTING TARGET OPTIONS

Next to the drop-down list for machines is a settings button (gear wheel). Pressing this button enters an editor in which JSON can be entered. If you are not familiar with JSON there are some resources listed in Demo Mode section below.

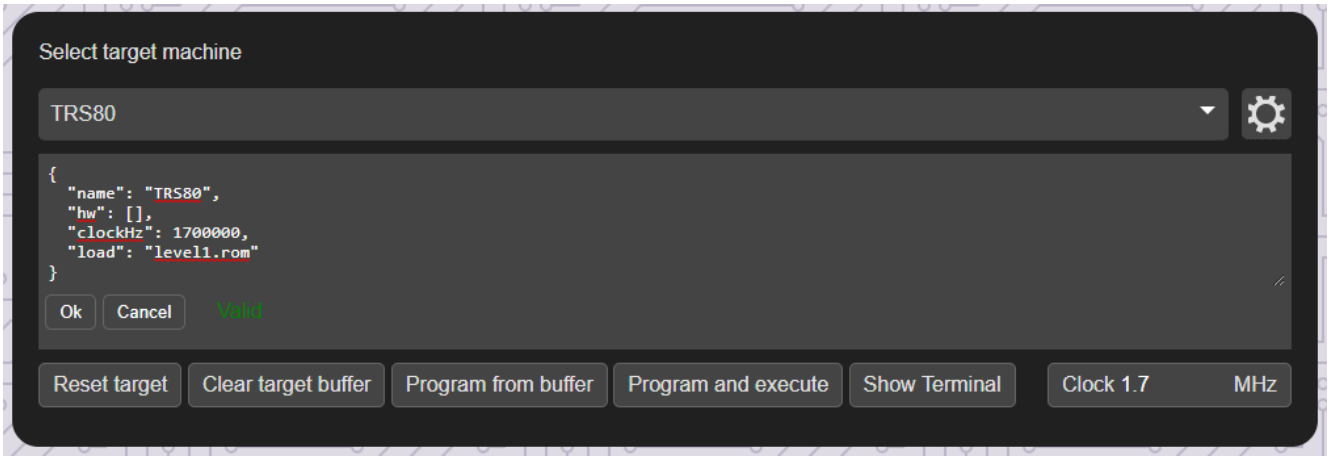
Firstly here's an example for a Serial Terminal:



The "name" should always match the name of the machine in the drop-down list – hopefully the code I've inserted should ensure this is always the case but it is best not to change it as things could become confused. Options include:

- "emulated6850" is specific to Serial Terminal machines. It enables software that monitors the IO accesses of the Z80 to IO where the RC2014's 6850 hardware is found and performs the functions of the UART – thus obviating the need to actually have the UART in place (if it is only used for driving the serial terminal).
- The "hw" section is currently a place-holder. In future I envisage that hardware could be listed here and support for things like paged RAM boards could be handled.

The second example is for a Z80 target machine:



- "clockHz" can be set on any machine (including Serial Terminal) and controls the initial clock speed generated
- "load" allows a file to be specified which will be loaded on startup

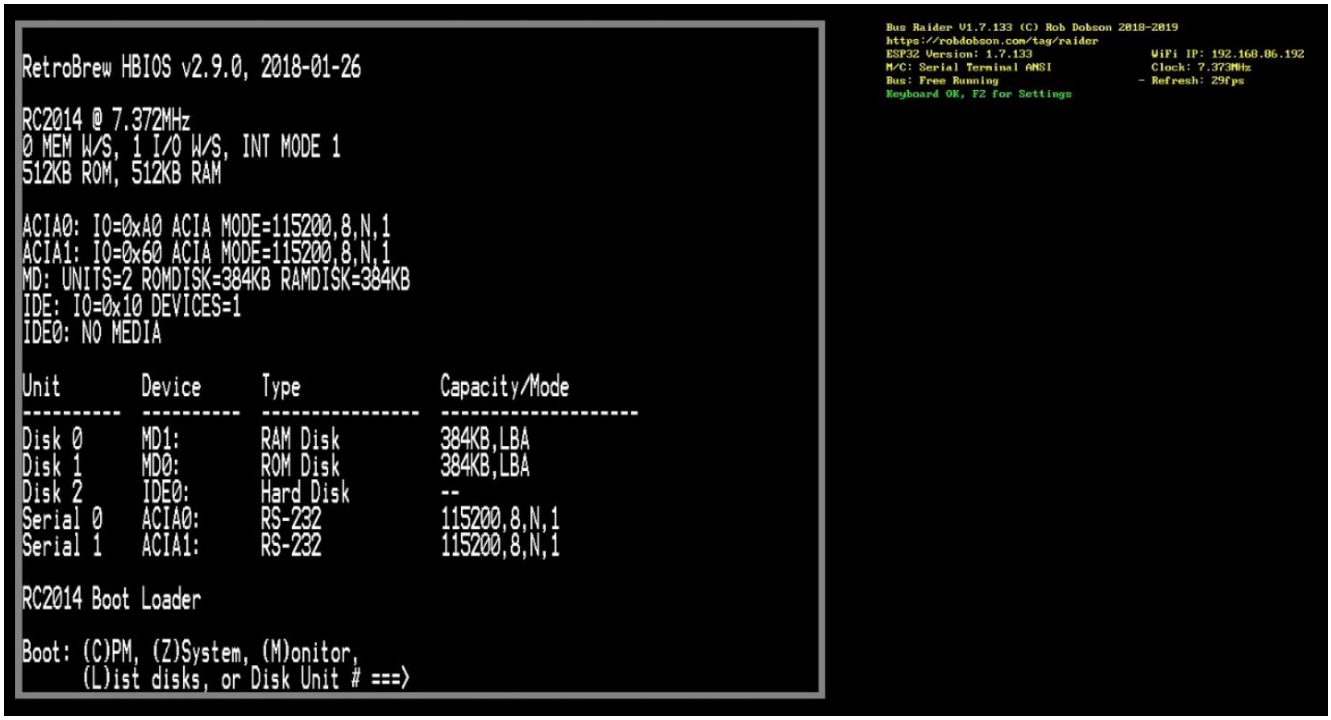
When a machine is selected these settings will automatically be applied and they are stored in a hidden file.

SERIAL TERMINAL MACHINES

There are several methods for using the BusRaider to communicate with a "serial terminal" machine. Any of these methods can make use of a real UART (68B50 or SIO) if the jumpers are in place to allow the BusRaider to use that UART (see Appendix 8). Alternatively the emulated 6850 can be used and this is enabled as described in the Selecting Target Options section above.

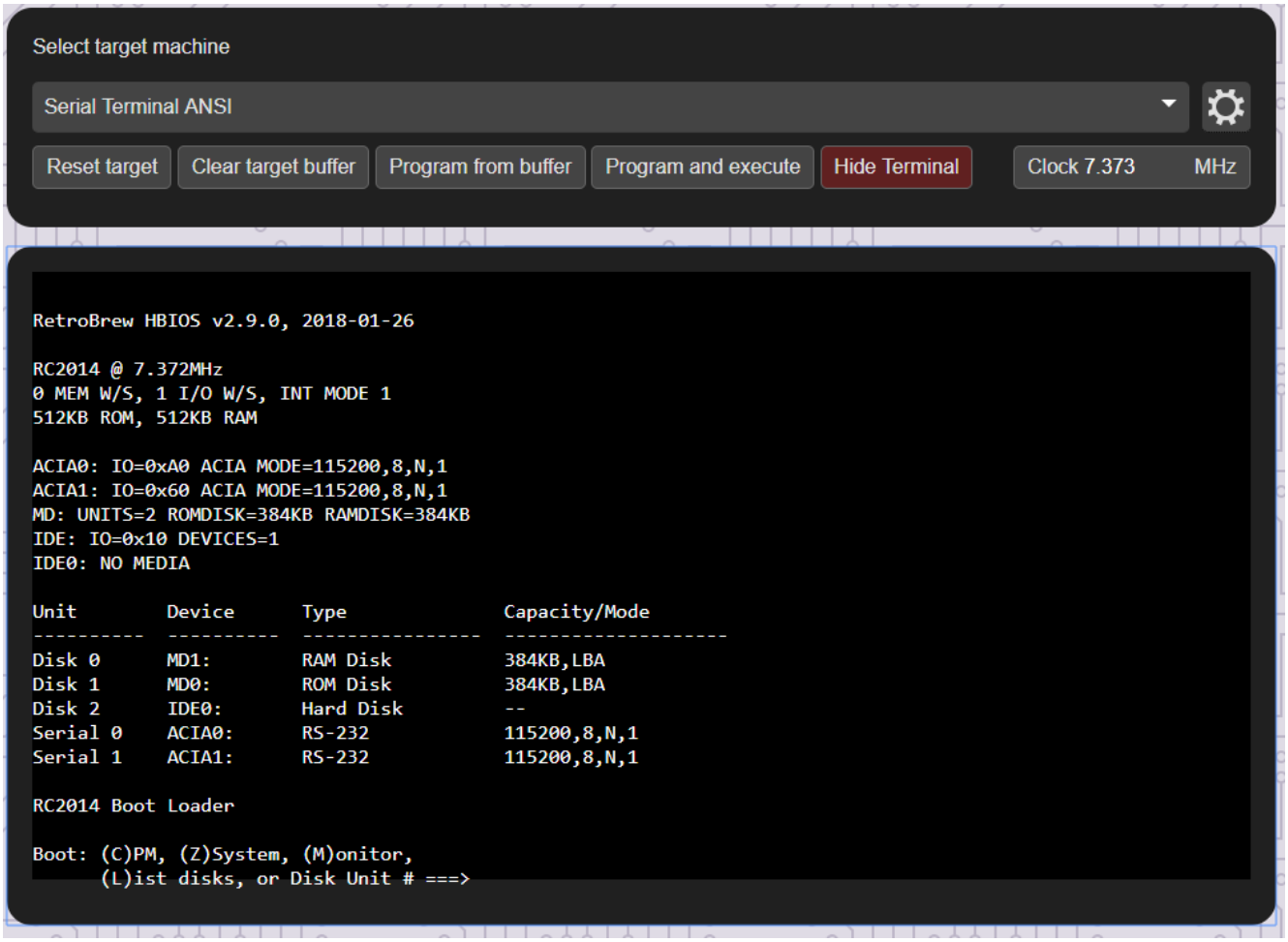
Using the HDMI Screen and USB Keyboard attached to the Pi Zero

The most obvious method for interacting with the serial terminal is to use the screen and keyboard attached to the Pi Zero. This should update quickly and give good emulation of either ANSI or H19 terminal escape codes depending on which you have selected from the drop-down list.



Using the Web UI Serial Terminal Mirror

Once the serial channel is enabled pressing the Show Terminal button in the Web UI mirrors the serial terminal output to the Web UI. Here you can interact with the machine using the standard keyboard of the machine with the Web UI. You can also interact using the USB keyboard.

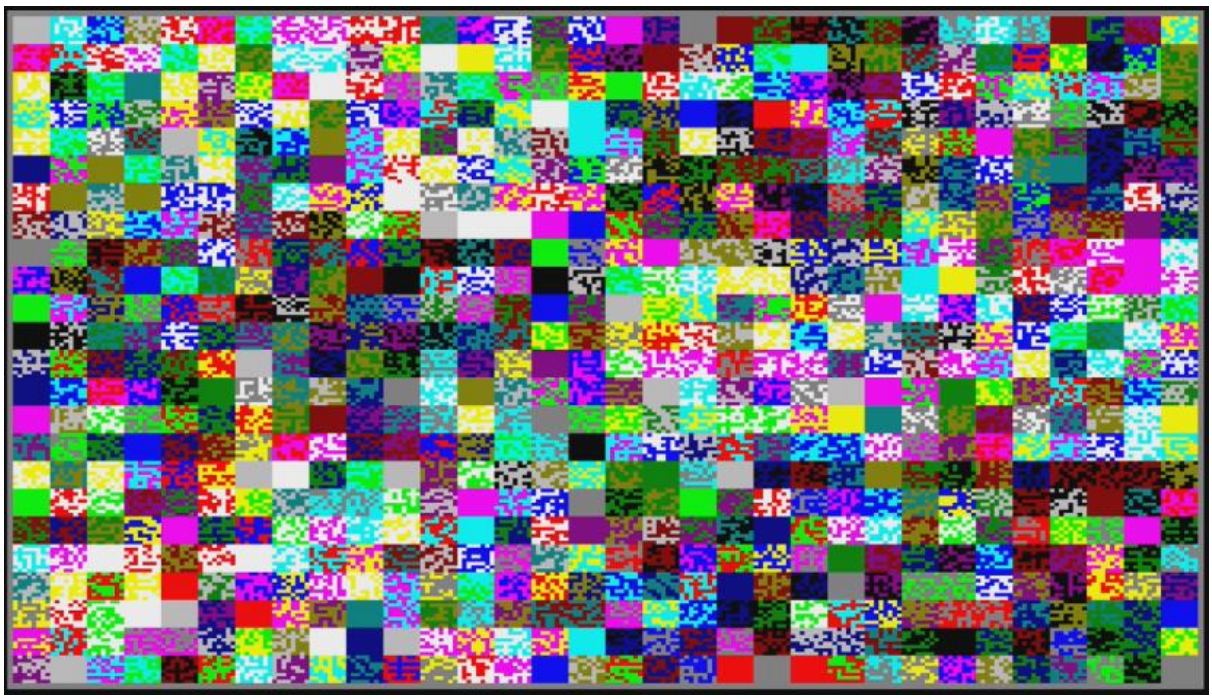


Telnet connection to Serial Terminal

A further option is to use telnet to connect to the serial terminal. Currently this is only working with a genuine UART and not with the emulated one. To connect use either the command line telnet in Linux/Mac or a program such as Putty on Windows.

MEMORY MAPPED DISPLAY MACHINES

Once you have made your selection of a memory-mapped display machine you should see the HDMI display showing the output from your chosen machine. At this stage it is expected to be random garbage that was in the memory of the Z80 at power up, so something like one of these displays:



For the ZX Spectrum or the following for the TRS80:

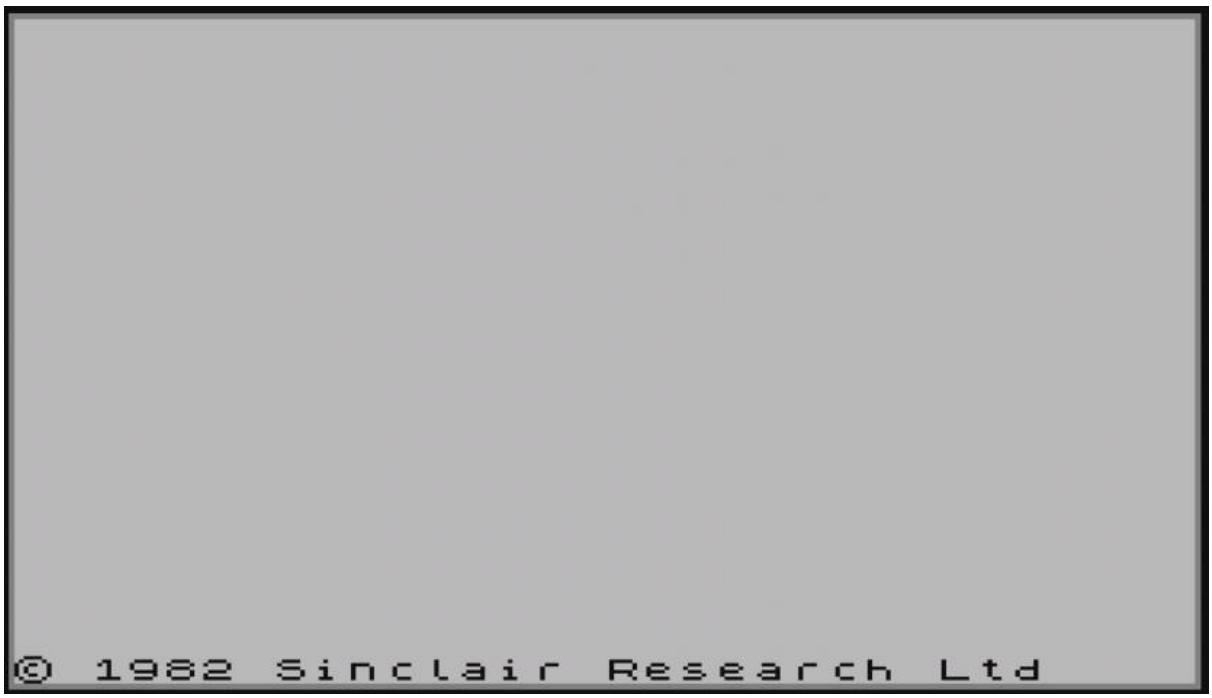
RUNNING A BASE ROM IMAGE

Now you are ready to run some software. First locate a ROM image file suitable for the machine you have chosen.

- If you have selected the ZX Spectrum then you can find a suitable ROM image here http://www.shadowmagic.org.uk/spectrum/roms.html and, specifically, the 48.rom file is the one to choose initially.
- For the TRS80 the there are some ROM images here https://github.com/lkesteloot/trs80/tree/master/roms, and the file level1.rom works well

Download the file to the computer that you have the WebUI running on and then drag and drop the file onto the middle section of the UI – where it says “Drop files to immediately run on the target”.

You should see something like the following:



Or:

At this point, if you have a USB keyboard connected, you should be able to type characters and see them appear on the target machine's display. You can run simple BASIC programs like this, for instance:

RUNNING A PROGRAM

Running other software such as games is just the same as running a base ROM. You can drag and drop files or use the storage on the ESP32 as described later in this document.

For example to run the Galaxy Invasion game you will need to have the TRS80 machine selected and the level1.rom in place first (as described in the previous section).
Then I have used a file called galinvs from this list http://www.trs-80.com/processsqlsearch.php simply download the file (it is a ZIP), extract the contents and drop the first file in the ZIP (gal5b00a.cmd but I think any one of them will do as they all seem to be the same) on the "Drop files to immediately run on the target" section of the WebUI.

Note that for the TRS80 the F1 key is used for CLEAR. This is needed to start the Galaxy Invasion game.
When running a program in this way the program is placed directly into memory using the BUSRQ/BUSACK mechanism described in Appendix 3 and execution is performed using the method described in Appendix 5.

CONTROLLING THE TARGET

There are a number of buttons in "Select target machine" section of the WebUI which are used to control the target computer. Before explaining the function of these it is necessary to explain that the BusRaider maintains a block-based record of data (and programs) sent to it that are destined for the target machine. This is referred to as the Target Buffer. When we dropped files into the "Drop files to immediately run" area this target buffer was used temporarily but then cleared once the file had been sent to the target and executed. There are a number of functions of the BusRaider which give finer grained control of the target buffer and some of these are accessed through the buttons described in this section.

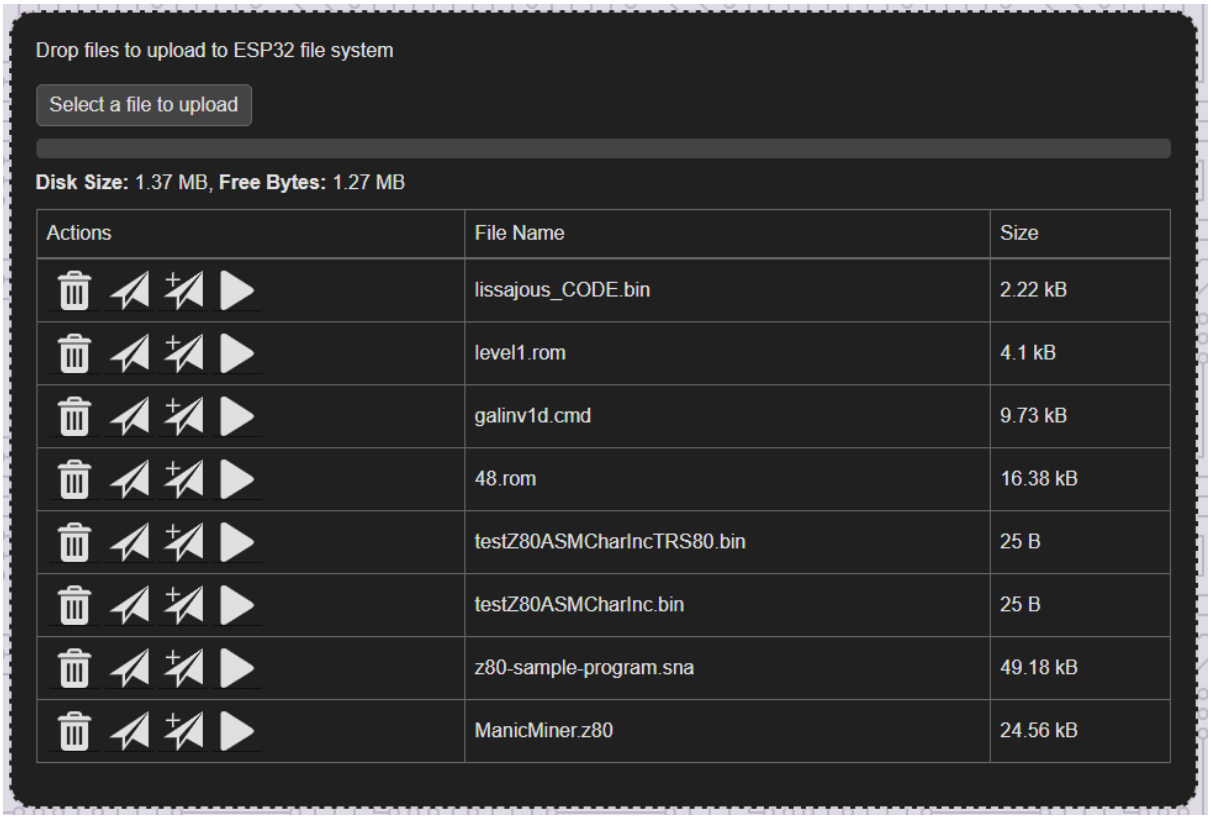
- Reset target. As expected this sends a Reset signal to the target machine. It also resets the bus controller so that wait-state generation is cleared.
- Clear target buffer. Clears the target buffer. This has no immediate impact on the target machine but any data or programs currently in the target buffer are removed.
- Program from buffer. This sends all the data in the target buffer to the target machine (using the BUSRQ/BUSACK mechanism described in Appendix 3. It then clears the target buffer.
- Program and execute. As Program from buffer but also issues a RESET to the target machine once done.

The use of the latter three buttons may become clearer in the next section.

STORING PROGRAMS TO RUN LATER

To store a program for later use simply drag and drop the file in the section labelled "Drop files to upload to ESP32 file system"
Currently all programs uploaded in this way are stored in the internal SPIFFS flash memory of the ESP32 which is limited to around 1.3MB. There is also an SD card slot on the BusRaider and I plan that this will be operational at some point. When it is the SD card (when inserted) will take precedence over the SPIFFS file system. So, once this is

supported, when an SD card is inserted on power up (dynamic insertion won't be supported) then the files listed in the UI will be the files on the SD card and any files on the SPIFFS area will not be shown.
For each file uploaded to the ESP32 file system a number of actions are available:



- The garbage can is obvious (I hope) and just deletes the file.
- The paper plane symbol sends this file to the target buffer. As described in the previous section sending to the target buffer does not immediately affect the target machine. The file is simply in a holding area where the blocks of data and program are stored waiting to be sent to the target machine.
- The paper plane with + sign appends the file to the target buffer – and again it does not yet affect the target machine.
- The play icon (triangle) causes the file to be run immediately – this uses the target buffer in the same way that an immediate upload does.

Perhaps it is now necessary to explain why the target buffer exists. The reason is that I found it generally useful to build up an image of the program I wanted to execute in more than one stage. Perhaps you would like to run a game which relies on the base ROM of the target machine for instance. You could just run the ROM and then run the game. But the target buffer allows you to assemble the ROM and game together and then execute the whole thing in one go. Not completely essential I guess but I have found it of some use. The functionality is also used in the Demo Mode (described later in this document).

DEMO MODE

The BusRaider supports a demonstration (Demo) mode as follows:

- A file called demo.json must be present on the SPIFFS file system (this will be true for the SD Card system instead once implemented and assuming an SD Card is installed in the BusRaider)
- The demo.json file must have contents following the description and example below

- When the DEMO button is pressed (this is below the ESP32 RESET button to the right of the ESP32 when looking at the component side), the BusRaider will rotate through each of the options described in the demo file (one per click of the button).

The format of the demo.json file is JSON (Javascript Object Notation) https://en.wikipedia.org/wiki/JSON

The file must contain a list called “demo” which is a list of objects, each one describing one demonstration case.

The demonstration case object can have the following name/value pairs:

- “file”: the name of the file (which must be on the ESP32 file system) to execute in this demonstration case
- “mc”: the name of the machine to be run the demonstration case on
- “preload”: a list of files which need to also be loaded into the target machine in order for the demonstration case to run properly

For example here is a Demo Mode file which runs a number of programs on different machines. Each time the Demo button is pressed a different game or system ROM prompt will be started on the specified machine.

```
{"demo":[
    {"file":"level1.rom", "mc":"TRS80"},
    {"file":"galinv1d.cmd", "mc":"TRS80", "preload":["level1.rom"]}
    {"file":"48.rom", "mc":"ZX Spectrum"},
    {"file":"lissajous.rom", "mc":"Rob's Z80"}
]}
```

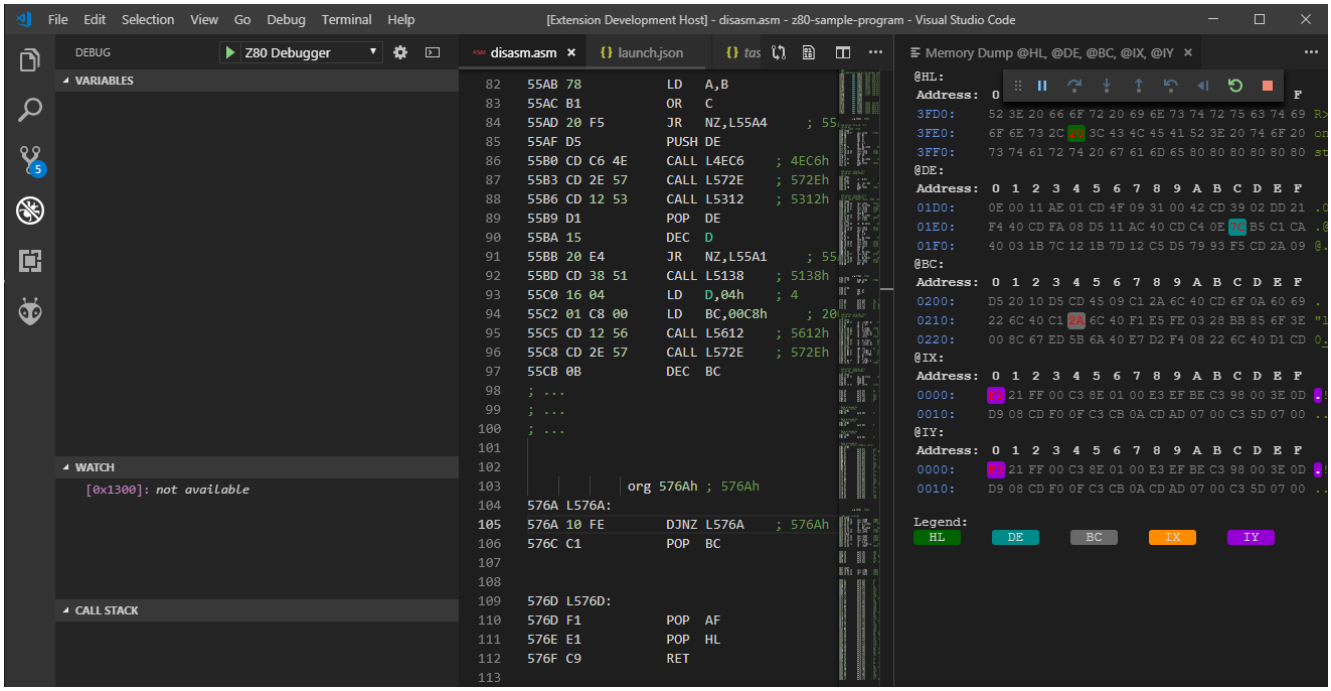
DEBUGGING A PROGRAM

Before debugging a program you will need to install Visual Studio Code https://code.visualstudio.com/ and follow the instructions to set up the Z80 Debug add-on as explained here https://github.com/maziac/z80-debug. Note that you do not need to install ZEsarUX as indicated in the notes as we will be using BusRaider in place of ZEsarUX.
The next step is to follow instructions (https://github.com/maziac/z80-debug/blob/master/documentation/Usage.md) for creating a launch.json file in the .vscode folder of a new project. Looking at the typical file in the Usage.md document you can see that there is a name/value pair called zhostname – highlighted in yellow below. This determines the device that the debugger will connect to and, in our case, it should be the BusRaider. In the example below I've chosen to connect to the IP address 192.168.86.192 which is my busraider's IP address. You should see yours in the yellow text status box on the HDMI output from the Pi Zero.

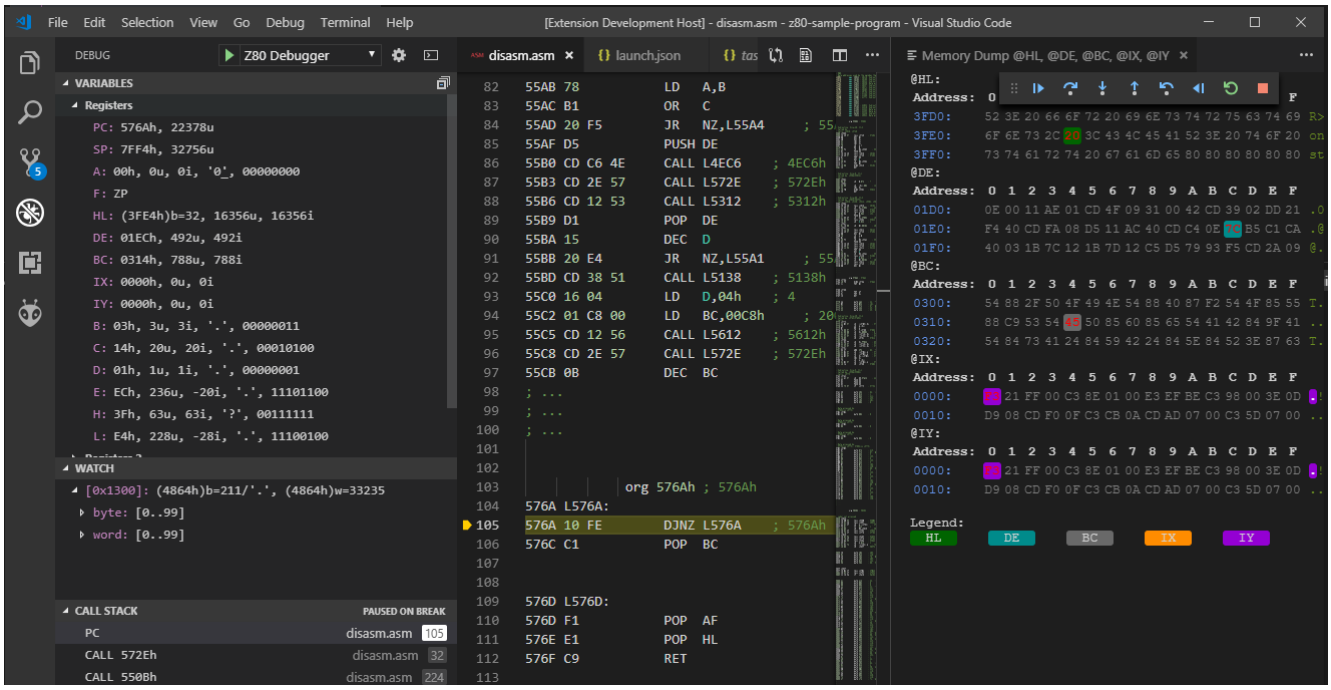
```
"configurations": [
        {
            "type": "z80-debug",
            "request": "launch",
            "name": "Z80 Debugger",
            "zhostname": "192.168.86.192",
            "zport": 10000,
            "listFiles": [
                // "../rom48.list",
                {
                    "path": "z80-sample-program.list",
                    "useFiles": true,
                    "asm": "sjasmplus",
                    "mainFile": "main.asm"
                }
            ],
            "startAutomatically": false,
            "skipInterrupt": true,
            "commandsAfterLaunch": [
                //"-sprites",
                //"-patterns"
            ],
            "disassemblerArgs": {
                "esxdosRst": true
            },
            "rootFolder": "${workspaceFolder}",
            "topOfStack": "stack_top",
            "load": "z80-sample-program.sna",
            "smallValuesMaximum": 513,
            "tmpDir": ".tmp"
       }
```

The settings in this file are described in the Usage.md document and the most important are probably the listFiles and load settings. The contents of the load setting are currently sent to the BusRaider but, right now, they are ignored and the currently running program is debugged so it is necessary to ensure that this corresponds to the ones you setup the list file for.
The other values in the launch.json file should be set as described in the Usage.md document linked above. The support for debugging is still at an early stage and I haven't worked out what a lot of it does myself. Some functionality will probably not work at all so we are all in new territory!

When you get Z80 Debug running – using the launch configuration above and selecting this by clicking the “bug” symbol in the left-most toolbar and the run button next to configuration Z80 Debug. You should see something like this:



And when you the press the pause button (in the toolbar towards the top-right – although yours might be elsewhere as it is floating) you should see something like this:



Due to the way the pause mechanism works it may take an additional step (down-arrow) to get the registers, memory and disassembly in synch.

The registers and RAM are captured using techniques described in the Appendix 5.3.

DIRECT BUS CONTROL

Control of the bus is available via a specific section of the REST API (over WiFi). This allows direct control over bus actions such as setting the address bus, setting/getting the data bus, setting processor control lines High/Low at will. This functionality is particularly useful for setting up tests of hardware under development.

The API can be accessed via the regular WiFi connection and can even be used directly from a browser address bar. For example:

```
http://busraideripaddr/targetcmd/rawBusControlOn
```

Where: busraiderapiaddr is the IP Address of your BusRaider.

The response should be something like:

```
{"cmdName":"rawBusControlOnResp","err":"ok","raw":"3fe6d033","pib":"fe","ctrl":"MIRW.","msgIdx":0,"dataLen":0}
```

The important part here is the “err”:”ok” which indicates the command completed successfully. The additional information isn’t relevant right now but is explained later on.

The commands available are as follows:

- rawBusControlOn / rawBusControlOff
- rawBusWaitDisable / rawBusWaitClear
- rawBusTake / rawBusRelease
- rawBusClockEnable / rawBusClockDisable
- rawBusSetAddress / rawBusSetData
- rawBusSetLine
- rawBusGetData

As an example, here is a Python script to exercise the bus:

```
# Import the requests library
import requests

url = "http://busraideripaddr/targetcmd/"
# Disable BusRaider's managed control of the bus
req = requests.request("get", url + "rawBusControlOn")
print(req.json())
# Disable and clear WAIT state generation
req = requests.request("get", url + "rawBusWaitDisable")
req = requests.request("get", url + "rawBusWaitClear")
# Take control of the bus
req = requests.request("get", url + "rawBusTake")
# Set the address bus to 1234 hexadecimal
req = requests.request("get", url + "rawBusSetAddress/1234")
# Set the data bus to 55 hexadecimal
req = requests.request("get", url + "rawBusSetData/55")
# Write to memory at this address
req = requests.request("get", url + "rawBusSetLine/MREQ/0")
req = requests.request("get", url + "rawBusSetLine/WR/0")
req = requests.request("get", url + "rawBusSetLine/WR/1")
req = requests.request("get", url + "rawBusSetLine/MREQ/1")
# Note that BUSRQ isn't released or control returned to BusRaider
# This is to ensure the test condition can be viewed as required
# To restore control either use rawBusRelease and rawBusControlOff
# Or just reset the system
```

APPENDIX 1: UPDATING THE ESP32 FIRMWARE

The ESP32 firmware is contained in the folder BusRaiderESP32 in the GitHub repository. To build and run this code you will need the following:

- Visual Studio Code (https://code.visualstudio.com/) – referred to sometimes as VSCode
- PlatformIO - this can be installed from VSCode by clicking the Extensions icon in the left-hand column (it looks like a square within a square) and searching for PlatformIO - more details at https://platformio.org/
- Espressif 32 Platform - in the Home Page of PlatformIO (click the house icon in the bottom-left toolbar to get the PlatformIO Home Page) select Platforms and search for Espressif 32 then install it
- Source code for the entire project – from https://github.com/robdobsn/PiBusRaider - the simplest way to get this code is either just to download the ZIP file using the green Clone or download button or clone it using the git command line or GitHub desktop tools (https://desktop.github.com/)

Optionally you will also need:

- USB to Serial adapter as described in Appendix 9. This is optional because you can program the ESP32 over the air (OTA) if you have WiFi set up and are comfortable sending a curl command to transfer the program (more details below).
- The curl program https://curl.haxx.se/download.html which is needed if you are using OTA update

Open the folder EspSw/BusRaiderEs32 in Visual Studio Code (the PlatformIO home page will probably launch). You should see folders lib and src and the file main.cpp in the project explorer panel. If you don't you have probably opened a folder one up or down in the tree.

At this point you should be able to open the folder for the BusRaiderESP32 sub-project (it won't work if you open the parent folder for the whole of BusRaider) in VSCode and then click the tick icon in the bottom-left toolbar. This should build the BusRaiderESP32 project and you should see something like this in VSCode.

```
Compiling .pioenvs\featheresp32\libd12\RdRestAPISystem\RestAPISystem.cpp.o
Archiving .pioenvs\featheresp32\libd12\libRdRestAPISystem.a
Indexing .pioenvs\featheresp32\libd12\libRdRestAPISystem.a
Linking .pioenvs\featheresp32\firmware.elf
Building .pioenvs\featheresp32\firmware.bin
Retrieving maximum program size .pioenvs\featheresp32\firmware.elf
Checking size .pioenvs\featheresp32\firmware.elf
Memory Usage -> http://bit.ly/pio-memory-usage
DATA:    [==        ]  15.0% (used 49216 bytes from 327680 bytes)
PROGRAM: [========  ]  84.4% (used 1106418 bytes from 1310720 bytes)
```

Appendix 1.1 Programming the ESP32

There are two ways to reprogram the ESP32:

- OTA (over-the-air) update capability that I have built into the existing firmware. This is probably the simplest option but it requires that my firmware is already on the ESP32, is running and is connected to WiFi – the connection LED will be blinking.
- Using a USB to Serial cable (FTDI or similar) as described in Appendix 9. This will work in all cases (even if the ESP32 is a new one that hasn't been programmed before) but requires that you first put the ESP32 into download mode.

Appendix 1.1.1 OTA Update of ESP32 Firmware

When you built the new firmware for the ESP32 a binary file was created in the following location (inside the folder v which is part of the project you downloaded from GitHub):

`.pioenvs/featheresp32/firmware.bin`

This is the firmware image that we need to transfer to the ESP32. Open a terminal or command prompt (you can use the terminal built into VSCode) and make sure you are in the base folder for the ESP32 project (EspSw/BusRaiderEsp32). The command to make this transfer is:

curl -F "data=@./.pioenvs/featheresp32/firmware.bin" http://busraiderip/espFirmwareUpdate

where:

- busraiderip is the IP address of your BusRaider (see the HDMI output yellow box for this)

What is happening here is that curl is issuing an http protocol request to the BusRaider over WiFi to access a Web API called espFirmwareUpdate. The ESP32 is programmed to respond to this by accepting the payload (which will be the firmware.bin file) and using the contents to reprogram itself.

Appendix 1.1.2 Updating ESP32 Firmware using USB to Serial

To reprogram the ESP32 over serial connection, first connect the serial cable (FTDI or similar) to the BusRaider as described in Appendix 9.

Next check the contents of the platformio.ini at the root of the ESP32/BusRaiderEsp32 folder. The most important setting is upload_port (which may or may not be present). On a Windows machine this setting should be the COM port (e.g. COM3) that the FTDI serial port is using (you can find this out with a terminal emulator program like TeraTerm (https://ttssh2.osdn.jp/index.html.en) or through the device manager in windows. On Linux it will probably be /dev/ttyUSB0. If you only have one serial connection you can remove the upload_port setting in platformIO.ini and it should locate the correct port for you.

Next hold down the DEMO button on the BusRaider down while pressing and releasing the ESP32 RESET button. If you have trouble with this then try connecting an FTDI cable and use a terminal emulator to see what is happening. When you get the sequence of button presses right you should see a message (see below) saying that the ESP32 is waiting to download.

At this point you MUST close any terminal emulator connections you may have or it won't be possible for PlatformIO to communicate with the ESP32.

```
rst:0x1 (POWERON_RESET),boot:0x3 (DOWNLOAD_BOOT(UART0/UART1/SDIO_REI_REO_V2))
waiting for download
```

Finally press the right-arrow button on the bottom-right toolbar in PlatformIO to start programming the ESP32. If all goes well, you should see the stages of the programming process and a success message.

```
esptool.py v2.6
Configuring upload protocol...
AVAILABLE: esp-prog, esptool, iot-bus-jtag, jlink, minimodule, olimex-arm-usb-ocd, olimex-arm-usb-ocd-h, olimex-
arm-usb-tiny-h, olimex-jtag-tiny, tumpa
CURRENT: upload_protocol = esptool
Looking for upload port...
Use manually specified: COM7
Uploading .pioenvs\featheresp32\firmware.bin
esptool.py v2.6
Serial port COM7
Connecting........___
Chip is ESP32D0WDQ6 (revision 1)
Features: WiFi, BT, Dual Core, Coding Scheme None
MAC: 30:ae:a4:16:96:28
Uploading stub...
Running stub...
Stub running...
Changing baud rate to 921600
Changed.
Configuring flash size...
Auto-detected Flash size: 4MB
Compressed 16176 bytes to 10658...
Wrote 16176 bytes (10658 compressed) at 0x00001000 in 0.1 seconds (effective 1011.0 kbit/s)...
Hash of data verified.
Compressed 3072 bytes to 144...
```

```
Wrote 3072 bytes (144 compressed) at 0x00008000 in 0.0 seconds (effective 1536.0 kbit/s)...
Hash of data verified.
Compressed 8192 bytes to 47...
Wrote 8192 bytes (47 compressed) at 0x0000e000 in 0.0 seconds (effective 4096.1 kbit/s)...
Hash of data verified.
Compressed 1106528 bytes to 593927...
Wrote 1106528 bytes (593927 compressed) at 0x00010000 in 10.0 seconds (effective 887.0 kbit/s)...
Hash of data verified.

Leaving...
Hard resetting via RTS pin...
```

NOTE: The ESP32 RESET button has to be pressed again when programming is finished to start the new ESP32 firmware.

APPENDIX 2: COPYING AND UPDATING THE PI SOFTWARE

The contents of the Raspberry Pi Zero (W) SD Card should be programmed from the folder PiSw/bin. There are four files:

- bootcode.bin and start.elf are the standard files that are always on the root of a Pi SD Card.
- kernel.img is the BusRaider software
- config.txt currently contains a command (disable_pvt=1) to disable a Pi feature that disables the main cpu for around 16uS on a regular basis.

Appendix 2.1 Building the Pi Software

To build the Pi software you will need:

- The CMake build system https://cmake.org/install/
- The GNU ARM toolchain https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads
- Source code for the entire project – from https://github.com/robdobsn/PiBusRaider - the simplest way to get this code is either just to download the ZIP file using the green Clone or download button or clone it using the git command line or GitHub desktop tools (https://desktop.github.com/)
- Optionally, the curl program https://curl.haxx.se/download.html which is needed if you are using OTA update

Open the folder PiSw in the project and execute the following commands:

Windows:

```
build.bat
```

or

```
build.bat c
```

the latter command cleans any cached files first – full rebuild.

Linux, etc:

```
cd ./build
rmdir -r CMakeFiles
rm CmakeCache.txt
cmake ../ -G"MinGW Makefiles" -DCMAKE_MAKE_PROGRAM=make -DCMAKE_TOOLCHAIN_FILE=./cmake/toolchain.cmake
make
```

After this you should have built the source code and the file kernel.img should be updated in the bin folder.

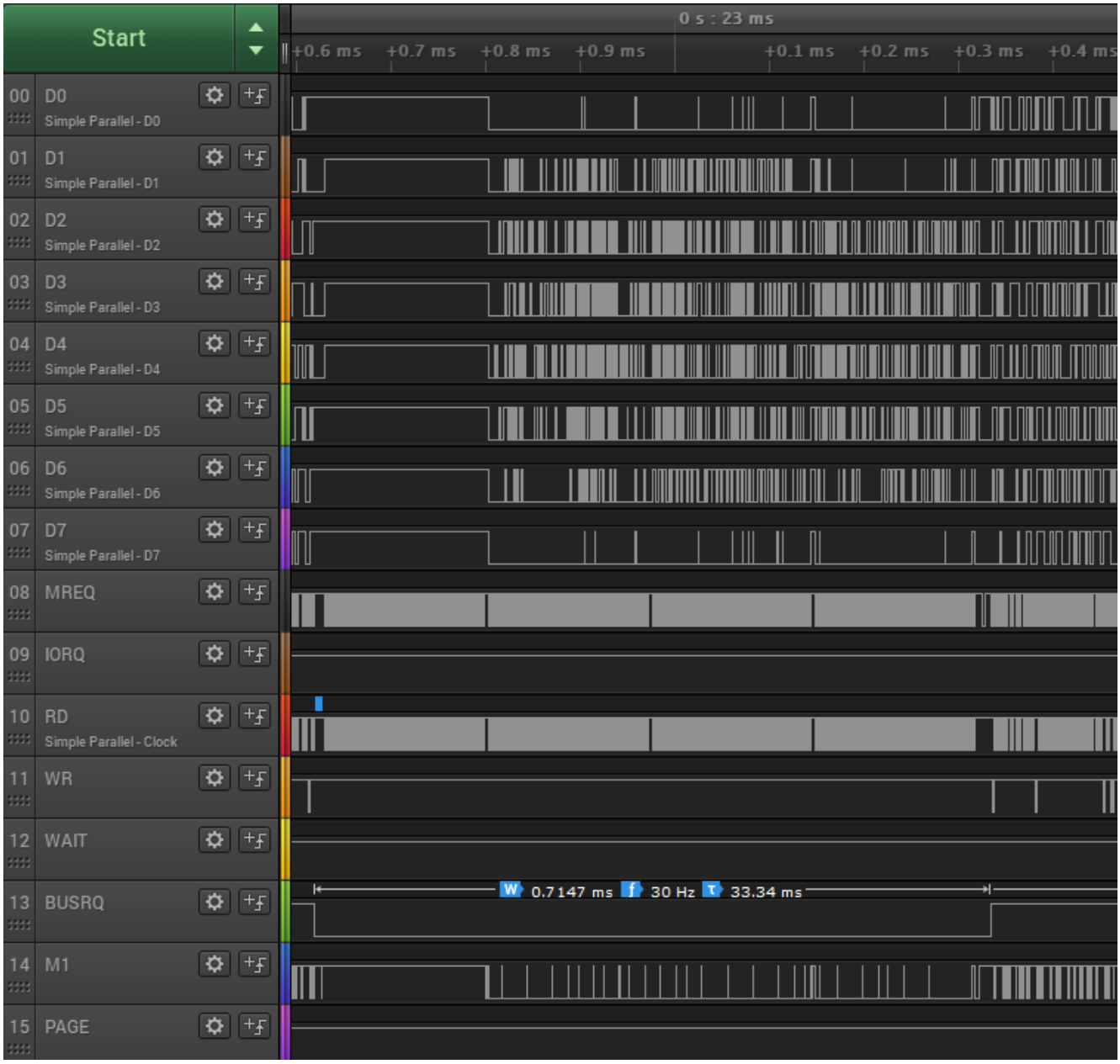
Now there are two options:

- To make the changes permanent, disconnect the power to the BusRaider and remove the MicroSD card from the Pi Zero (W). Place this MicroSD card into a reader on your PC and copy the new bin/kernel.img file over the one on the MicroSD card. Reinsert the card and power up.
- Using an OTA (over-the-air facility which enables quick but temporary (they will disappear after power cycle) changes to the Pi Software. To do this execute the following command from the PiSw folder in a terminal or command prompt:

```
curl -F "file=@bin/kernel.img" http://192.168.86.192/uploadpisw
```

APPENDIX 3: BUSRQ/BUSACK

When the BusRaider wants access to the target computer's memory it asserts the BUSRQ line and waits for a BUSACK in response. In the example below, which is a read of video RAM



We can see that the BUSRQ line is asserted by the BusRaider and shortly afterwards the Z80 responds with BUSACK to signal that the BusRaider has control of the bus. The BusRaider then accesses the graphics memory (you can see the MREQ and RD lines are the only ones active during this period) and the fact that the D0-D7 lines are generally 0 during this time indicates that the screen is partly blank. This is actually from a TRS80 emulation while the Galaxy Invasion program is running. Also, the video memory grab takes around 720uS in this case (of a 30Hz or 33ms refresh period) so is slowing the Z80 down by around 2.5%. For machines like the Spectrum which have larger display memory areas the effect is more significant.

APPENDIX 4 CURRENTLY DEPRECATED OPTIONS

In the previous Web UI of the BusRaider there were two options to set advanced features. These are currently deprecated but may come back in future versions:

- RAM emulation
- Opcode injection

APPENDIX 5: OPCODE INJECTION

One of the more interesting features of the BusRaider is its ability to insert opcodes (and associated data) onto the data bus of the target processor. This capability is used for two purposes:

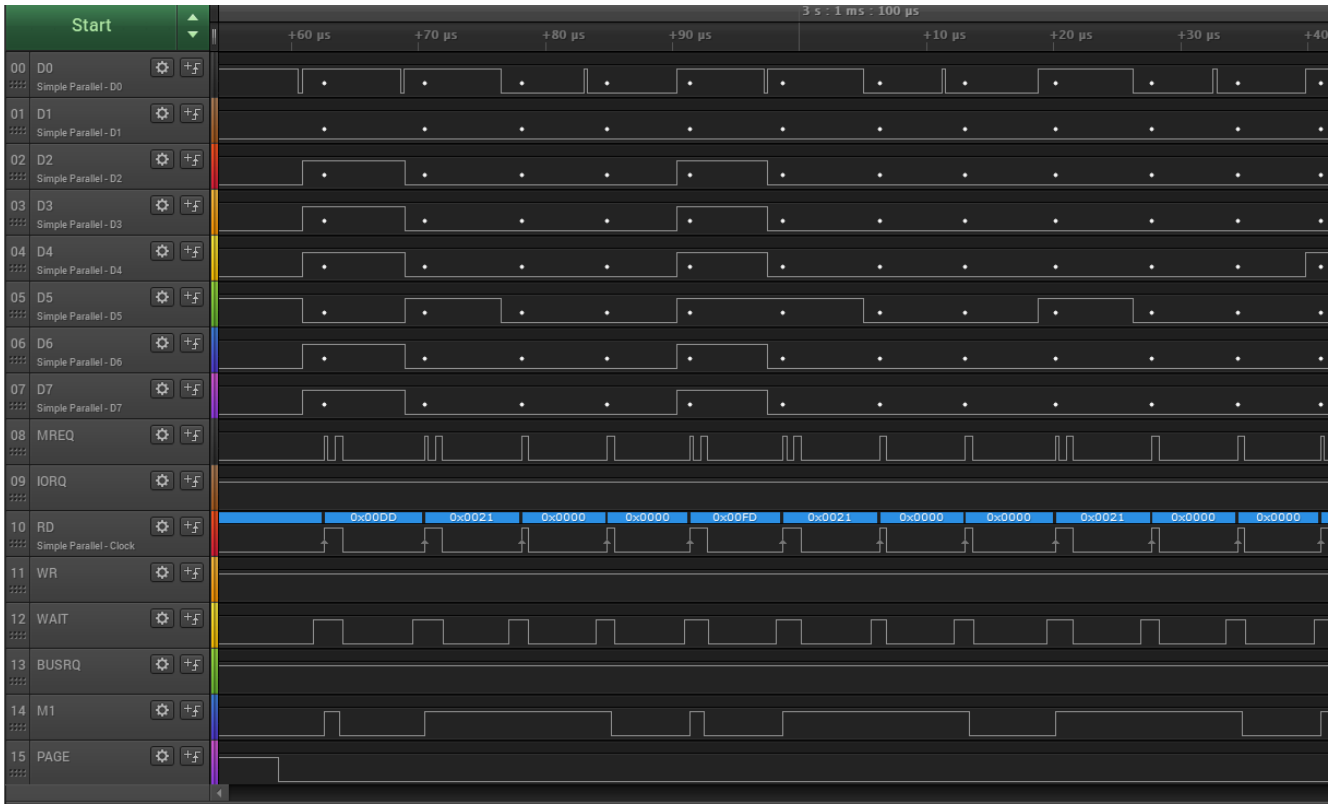
- To execute software downloaded via the BusRaider, for instance with file formats such a SNA (snapshot) format for the ZX Spectrum which contain register values and execution address.
- For single-step debugging to read the values of the target processor's registers at each step.

Operation in each of these cases depends on the use of wait-state generation to hold the target processor at the current execution point and then selectively releasing the wait-state while appropriate instructions are placed on the bus or data read from the bus as appropriate.

Appendix 5.1 Program Execution with Opcode Injection

NOTE: This option isn't available in the current BusRaider version and programs are always executed by inserting instructions into the RAM of the target machine (see Appendix 5.2). It will probably be restored in future.

In the case of target program execution with opcode injection, the bus control sequence looks as follows:



What we are seeing here is a view of the data bus and control bus while opcode insertion takes place for program execution. The actual instructions being injected are shown below. For each actual opcode (including prefixes 0xDD, 0xED, 0xFD and 0xCB) the M1 line is asserted (low) and the BusRaider uses this to synchronize the start of the sequence. You can see the PAGE line at the bottom which is active (low) when opcodes or data are being inserted

onto the Z80 bus. Looking at the blue line you should see the same opcodes and data as in the table below, starting after the leading 0x0000.

```
        0x00,                       // nop - in case previous instruction was prefixed
        0xdd, 0x21, 0x00, 0x00,     // ld ix, xxxx
        0xfd, 0x21, 0x00, 0x00,     // ld iy, xxxx
        0x21, 0x00, 0x00,           // ld hl, xxxx
        0x11, 0x00, 0x00,           // ld de, xxxx
        0x01, 0x00, 0x00,           // ld bc, xxxx
        0xd9,                       // exx
        0x21, 0x00, 0x00,           // ld hl, xxxx
        0x11, 0x00, 0x00,           // ld de, xxxx
        0x01, 0x00, 0x00,           // ld bc, xxxx
        0xf1, 0x00, 0x00,           // pop af + two bytes that are read as if from stack
        0x08,                       // ex af,af'
        0xf1, 0x00, 0x00,           // pop af + two bytes that are read as if from stack
        0x31, 0x00, 0x00,           // ld sp, xxxx
        0x3e, 0x00,                 // ld a, xx
        0xed, 0x47,                 // ld i, a
        0x3e, 0x00,                 // ld a, xx
        0xed, 0x4f,                 // ld r, a
        0x3e, 0x00,                 // ld a, xx
        0xed, 0x46,                 // im 0
        0xfb,                       // ei
        0xc3, 0x00, 0x00            // jp xxxx
```

The purpose of this code is to set the registers to be as they were when a snapshot was taken (normally this information is extracted from a snapshot file format such as the SNA, Z80 or ZTX formats commonly used with the ZX Spectrum). In cases where only the program counter (execution address) needs to be specified (e.g. TRS80 CMD format) the same opcode injection is used but the other registers are initialised with 0 values.

Appendix 5.2 Program Execution with Opcode Injection Off

When opcode injection is turned off (see appendix 4) the approach taken to program execution is to write code into two areas of the target machine's memory to set registers and execute the program on the target. The code written is as follows:

- Jump instruction at the reset vector location (0 for Z80) which makes execution jump to the register setting code
- Register setting code which is a block of around 55 bytes which is written to a "safe" area of RAM in the target machine. In general this "safe" area will be part of the display RAM and, in that case, the code may show up on the memory mapped display if it is not quickly overwritten.

The actual code executed in this case is identical to that which is injected to set registers as described in Appendix 5.1

Appendix 5.3 Register Query Bus Control

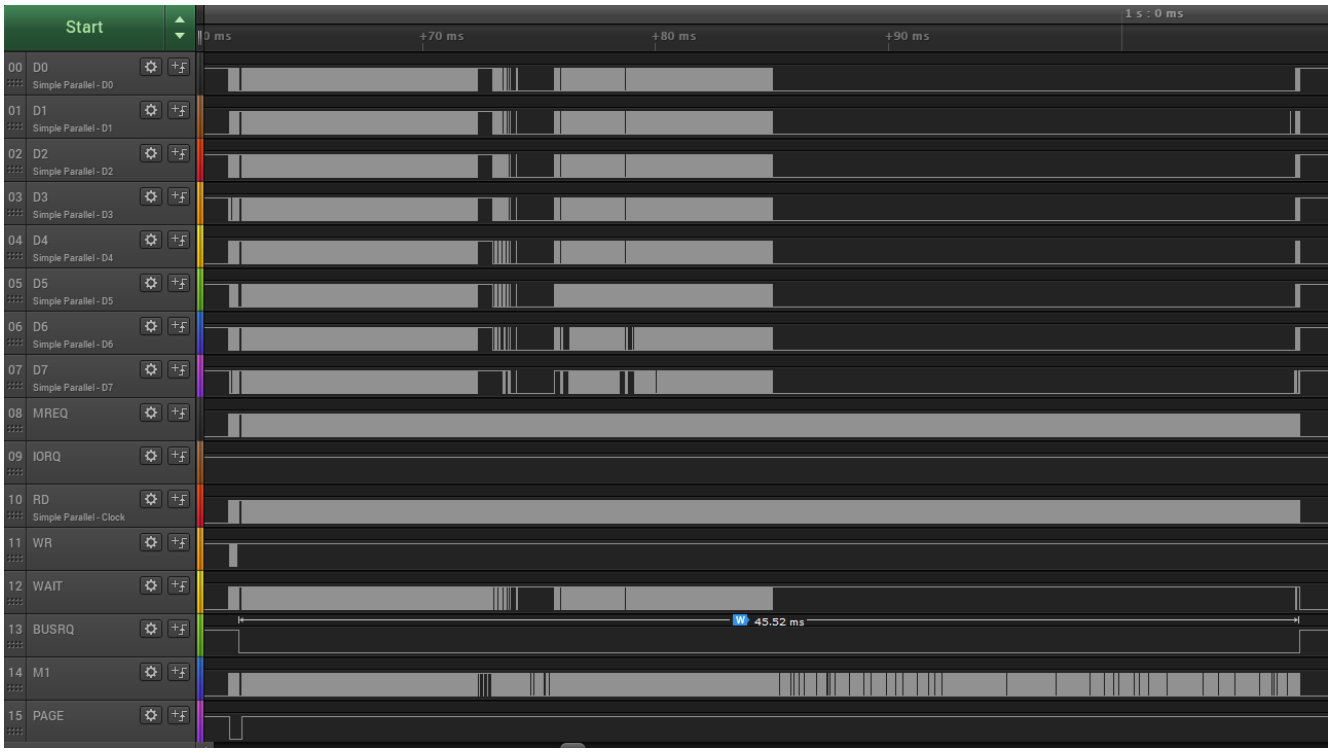
The register query opcode injection mechanism works in the same way as described above but with different opcodes and read operations. The overall bus control sequence for single-stepping (which is where register query is used) can be understood from a couple of levels and it is important to understand that this process begins and ends with the Z80 processor held in a wait-state.

Looking at the entirety of what happens when you click step in the debug UI:

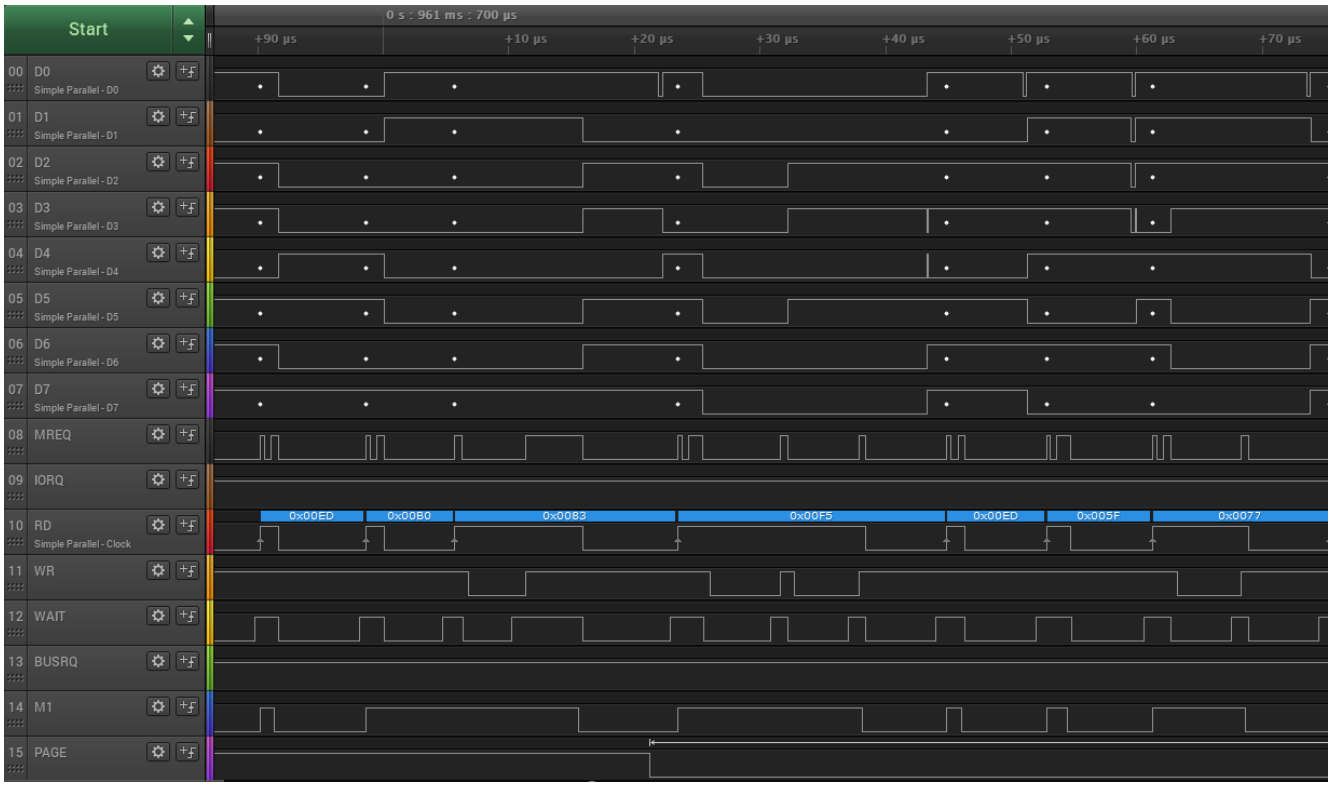
- Any instruction that is previously executing is allowed to complete and the start of a new M1 (instruction fetch) cycle (excluding an M1 on an instruction that was prefixed) is awaited
- When the new M1 cycle begins opcode insertion is started and, in place of the instruction the processor expected, a generated instruction is injected to query one of the registers and/or put the results onto the data bus.

- At certain points the data bus is read by the BusRaider and the contents are stored in its copy of the register state.
- This continues until all the register contents have been read.
- On next final opcode or data injection cycle the BUSRQ signal is generated (while the wait signal is still asserted)
- Wait is released
- BUSACK is awaited
- Once BUSACK is received the contents of target machine's entire memory is read and copied to a buffer in the Pi Zero
- The BUSRQ is released and the processor is freed to execute but with wait-state generation enabled on all memory requests and this results in it being held at the very next M1 cycle

The logic analyser capture below shows this entire activity including both the register get and the subsequent RAM grab.



Detail of the register read operation can be seen here:



The 0xED, 0xB0 and 0x83 values are the completion of the previous instruction (and LDIR prefixed instruction fetch with read and write following). Following this we can see the PAGE line being asserted to avoid RAM conflicts and the instructions 0xF5, 0xED, 0x5F, 0x77 being inserted. There is then a write cycle starting at around 62uS on the timeline and this is during execution of the 0x77 instruction which is LD (HL), A. At this point the BusRaider is reading the data bus for the information contained in this write operation (which happens to have been the R register contents a moment ago) and the BusRaider stores this in the register table that is returned to the debugger. The full set of opcodes inserted to query registers and then restore state are as follows:

```
                // loop:
        0xF5,           //      push af
        0xED, 0x5F,     //      ld a,r - note that this instruction puts IFF2 into PF flag
        0x77,           //      ld (hl),a
        0xED, 0x57,     //      ld a,i
        0x77,           //      ld (hl),a
        0x33,           //      inc sp
        0x33,           //      inc sp
        0x12,           //      ld (de),a
        0x02,           //      ld (bc),a
        0xD9,           //      exx
        0x77,           //      ld (hl),a
        0x12,           //      ld (de),a
        0x02,           //      ld (bc),a
        0xD9,           //      exx
        0x08,           //      ex af,af'
        0xF5,           //      push af
        0x33,           //      inc sp
        0x33,           //      inc sp
        0x08,           //      ex af,af'
        0xDD, 0xE5,     //      push ix
        0x33,           //      inc sp
        0x33,           //      inc sp
        0xFD, 0xE5,     //      push iy - no inc sp as we pop af lower down which restores sp to where it was
        0x3E, 0x00,     //      ld a, 0 - note that the value 0 gets changed in the code below
        0xED, 0x4F,     //      ld r, a
        0xF1, 0x00, 0x00,    //      pop af + two bytes that are read is if from stack
                        //      - note that the values 0, 0 get changed in the code below
        0x18, 0xDE      //      jr loop
```

The loop instruction at the end might be regarded as a bit of an oddity as it doesn't so much create a jump but just puts the program counter (PC) back to where it was before the register query code was injected.

APPENDIX 6 BRIEF HARDWARE OVERVIEW

The BusRaider hardware performs the following functions:

- Generation of BUSRQ signal to capture target bus
- Setting address and control bus and writing/reading data bus to effect target memory access during bus capture
- Wait-state generation on memory and/or IO access to hold processor during a read/write cycle
- Read access to address and control buses and read/write data bus to enable IO/memory system emulation, opcode injection and single-step debugging
- IRQ, NMI and RESET signal generation
- Processor clock generation to allow matching of the machine speed of a specific retro computer
- Paging signal generation to "page out" physical RAM when emulating memory or injecting opcodes
- Access to target bus serial IO to enable terminal emulation mode and telnet support

There are two processors, a bunch of 74 series hardware and some discrete components on the BusRaider which accomplish these functions:

- Raspberry Pi Zero (or Zero W) referred to elsewhere as Pi
- ESP32
- Address bus registers and drivers U3 and U4
- Address bus read drivers U9 and U11
- Data bus read/write driver U2
- Control bus read/write driver U1
- General bus logic and drivers U5, U8 and U14
- Wait state logic U10 and U13
- Multiplexer to increase Pi line controls U6
- Serial port hardware and jumpers
- Power supply regulator

There isn't too much complexity about the hardware. The two Flip-flops (U10 and U13) are perhaps the most complex aspect:

- 13 allows the Pi to enable, disable and clear MREQ and IORQ events that assert the WAIT line and cause Pi interrupts to handle the condition.
- U10 part 1 is used to keep the bus-driver for the data bus enabled until the end of a read cycle in the case where the Pi is supplying the data to be read.
- U10 part 2 ensures that RFSH cycles (an MREQ without RD or WR) are ignored and don't interrupt the processor.

U4 and U3 generate the address when in BUSRQ mode. U4 is a latching counter and driver, this was chosen to enable relatively quick incremental addressing on the lower 8 bus lines so a block of 256 bytes can be accessed relatively quickly in incremental fashion. U3 is a shift-register and driver, in this case the choice was made to allow an arbitrary address to be selected relatively quickly as this may be more commonly required on the upper bus lines (although in reality the choice of another counter would probably not have made much difference).

Appendix 7 Raspberry Pi Software

The Raspberry Pi Zero (W) operates in bare-metal mode (i.e. without an operating system) and is responsible for all real-time operations on the target machine's bus. It is responsible for:

- Configuring the hardware into the correct modes to operate as required for the user's settings (target machine, RAM emulation, paging, opcode injection, target clock frequency, etc).
- Generating the BUSRQ signal used to gain access to the target machine's program and memory-mapped graphics RAM. The rate at which this occurs is set in the descriptor-table for the machine being emulated – see Appendix 7.1 Machine Descriptors and Classes
- Handling the interrupts generated when an enabled wait-state occurs. This can be a memory or IO request and the interrupt behaviour is quite complex because different operating modes (RAM emulation, paging, opcode injection) are mainly handled in this activity – see Appendix 7.2 Wait-State Handling.
- Communicating with the ESP32 (and through that channel with the web-UI and/or Visual Studio add-on for single-step debugging). This is a bi-directional serial connection which currently operates at 500K baud.
- Generating the HDMI display signal for display on a connected monitor which is used to display the emulated graphics from the target machine and also used for diagnostic information, setting immediate-mode values such as WiFi SSID and password, etc.

Appendix 7.1 Machine Descriptors and Classes

The Raspberry Pi software is responsible for emulation of the target machine type (ZX Spectrum, TRS-80, etc). Each machine that can be emulated requires a separate C++ class to define its behaviour and there is also a table of settings (called the descriptor table) that each machine exposes to the Pi software.

APPENDIX 7.1.1 A QUICK NOTE ABOUT CLASSES

If you are familiar with classes (a part of object-oriented programming) then skip the following description but, if not, I hope the following will be a quick de-mystification of what I know is a testy topic for some people.

The idea of a class in C++ is actually very simple and this is an ideal example to describe its use. Imaging that you want to describe a number of different retro computers in a simple way to a stranger. One approach would involve "similarities and differences".

On the subject of similarities, for example, you could say that both a TRS-80 and a ZX-Spectrum use the Z80 processor and have some memory which is read-only, some which can be written to generate a text/graphics output and some which is just read/write for general program and data use.

Looking at the differences you could point to the different layout of graphics memory, the different (memory vs IO mapping of the keyboard matrix) and the Spectrum's use of a regular interrupt for timing.

To model this in software it makes sense to put the similar (common) parts into code which is shared and the differences into code which is different. And in an object oriented language like C++ this can be done easily using a base class and derived classes. So in the code you will see a file called McBase.h (and McBase.cpp) and this contains common code but, even more importantly, the descriptions of the interface functions that are used to "talk-to" the derived classes. Some of functions look like this:

```
    // Enable machine
    virtual void enable() = 0;

    // Disable machine
    virtual void disable() = 0;

    // Handle display refresh (called at a rate indicated by the machine's descriptor table)
    virtual void displayRefresh() = 0;
```

```
    // Handle a key press
    virtual void keyHandler(unsigned char ucModifiers, const unsigned char rawKeys[6]) = 0;

    // Handle a file
    virtual void fileHandler(const char* pFileInfo, const uint8_t* pFileData, int fileLen) = 0;
```

We have functions for enabling and disabling the machine (which are used when the machine is selected in the WebUI), for refreshing the display, handling a key and handling a file. All things that can readily be understood when we think about the differences between machines. The specific syntax around the virtual keyword and the = 0 at the end of each line results in the creation of a pure-virtual function which means that it must be overridden in the derived class otherwise the compiler complains. This is generally done to avoid unexpected functionality which is just due to forgetting to implement something.

APPENDIX 7.1.2 MACHINE DESCRIPTOR TABLE

The Descriptor Table is referred to elsewhere and this provides a default set of information about the machine so that standard behaviour can be generated by the Pi Zero code. An excerpt from the ZX Spectrum descriptor table is as follows:

```
McDescriptorTable McZXSpectrum::_descriptorTable = {
    // Machine name
    "ZX Spectrum",
    McDescriptorTable::PROCESSOR_Z80,
    // Required display refresh rate
    .displayRefreshRatePerSec = 50,
    .displayPixelsX = 256,
    .displayPixelsY = 192,
    .displayCellX = 8,
    .displayCellY = 8,
    .pixelScaleX = 4,
    .pixelScaleY = 3,
    .pFont = &__systemFont,
    .displayForeground = WGFX_WHITE,
    .displayBackground = WGFX_BLACK,
    // Clock
    .clockFrequencyHz = 3500000,
    // Interrupt rate per second
    .irqRate = 0,
    // Bus monitor
    .monitorIORQ = true,
    .monitorMREQ = false,
    .emulatedRAM = false,
    .emulatedRAMStart = 0,
    .emulatedRAMLen = 0,
    .setRegistersByInjection = false,
    .setRegistersCodeAddr = ZXSPECTRUM_DISP_RAM_ADDR
};
```

A lot of this is self explanatory I think but here are some notes on the less obvious values:

- The pixelScaleX and Y values determine how many Pi Zero pixels equal a Spectrum pixel.

- The irqRate isn't set in this case despite the fact that the Spectrum does require IRQ to be generated at the frame refresh rate. The reason for this is that the code synchronizes the actual grabbing of the display memory with the IRQ generation to ensure that they don't overlap which would cause issues (missed IRQs or IRQs generated when the BUSRQ signal was asserted).
- monitorMREQ is changed by the settings in the WebUI and whether debugging is in progress.
- emulatedRAMStart and emulatedRAMlength are used to determine if all of RAM should be emulated or just part of it. Currently only one area of emulated RAM (which may be the entirety of it) is supported and this must be set in the descriptor table (it isn't changeable through the UI).
- setRegistersByInjection is also changed by the WebUI settings
- setRegistersCodeAddress should be an area of safe memory that can be overwritten with register setting code – such as part of the graphics memory.

APPENDIX 8 SERIAL PORT JUMPERS

There are several sets of jumpers relating to serial connections. The serial ports which can be connected together in various combinations are as follows:

- ESP0
- ESP1
- ESP2
- Pi
- Z80Serial
- FTDI

For the Z80 there is also a separate set of jumpers which connects Z80#1 or Z80#2 (both from the RC2014 bus) to Z80Serial.

The standard configuration of the BusRaider is as follows:

- ESP0 <-> FTDI
- ESP1 <-> Pi
- ESP2 <-> Z80Serial

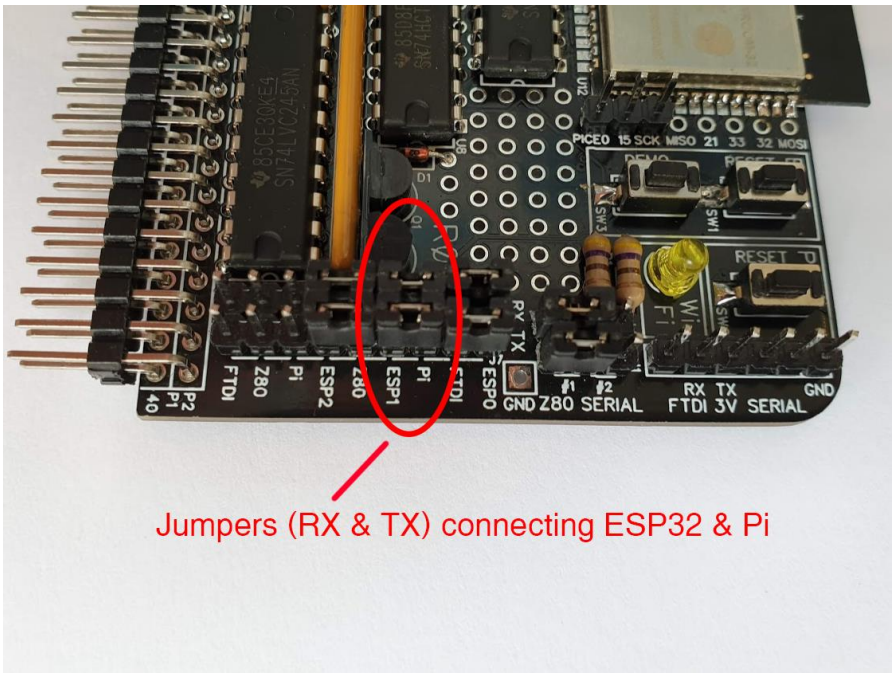
These are the connections shown and explained in the following sections.

Appendix 8.1 ESP0 to FTDI Jumpers

These are described in full in Appendix 9 FTDI Serial Cable Connector

Appendix 8.2 ESP32 to Pi Serial Jumpers

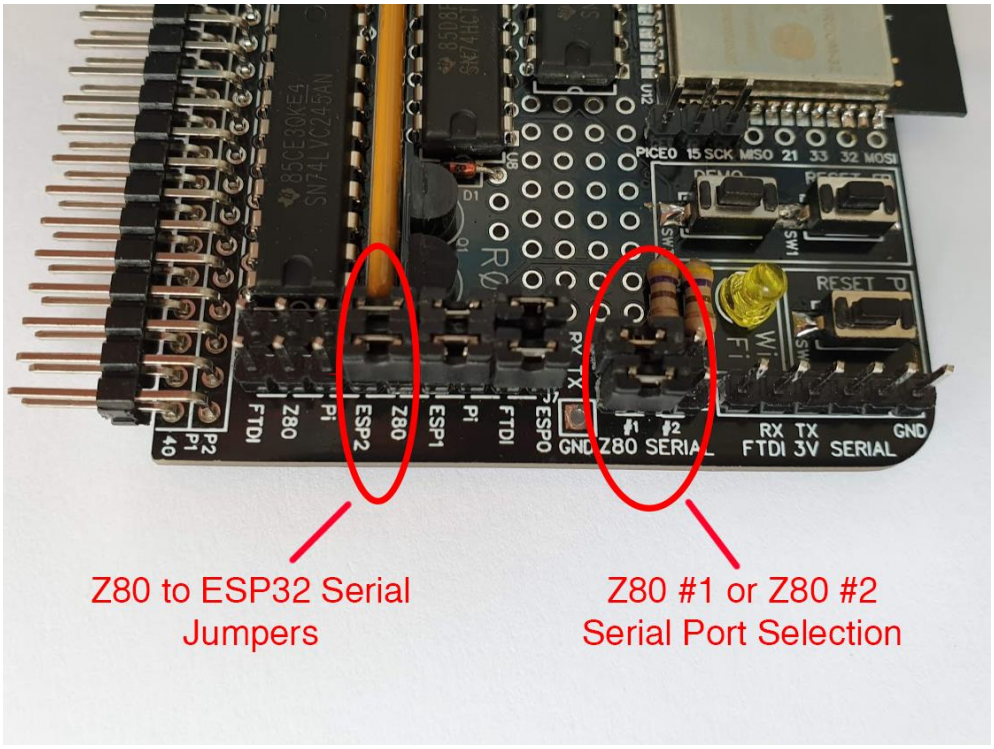
The jumpers shown below connect the ESP32 command port to the Pi serial port. Both jumpers need to be in place for full-duplex communication. If these jumpers are not in place the ESP32 and Pi will not be able to communicate.



Appendix 8.3 ESP32 to Z80 Jumpers and Z80 #1/#2 Selection

In order for the Z80 to communicate serially with the BusRaider (for telnet use or with the Serial Terminal machine) you will need to first select from Z80 #1 or Z80 #2 serial lines (from the RC2014 bus). This setting is chosen with the Z80 SERIAL jumpers that can be seen in the image below.

The jumpers shown to the left then connect the ESP32 port 2 (ESP2) to the selected Z80 serial port. Both sets of jumpers need to be in place for full-duplex communication



APPENDIX 9 FTDI SERIAL CABLE CONNECTOR

To set WiFi, program the ESP32 and see diagnostic messages it can help to use a 3V FTDI USB to serial cable and terminal emulator.

Suitable USB to Serial adapters are available on Amazon/Ebay such as 3V3 FTDI Cable https://www.amazon.co.uk/dp/B071WPW292 or equivalent "Prolific DebugCable" https://www.amazon.co.uk/gp/product/B01N4X3BJB

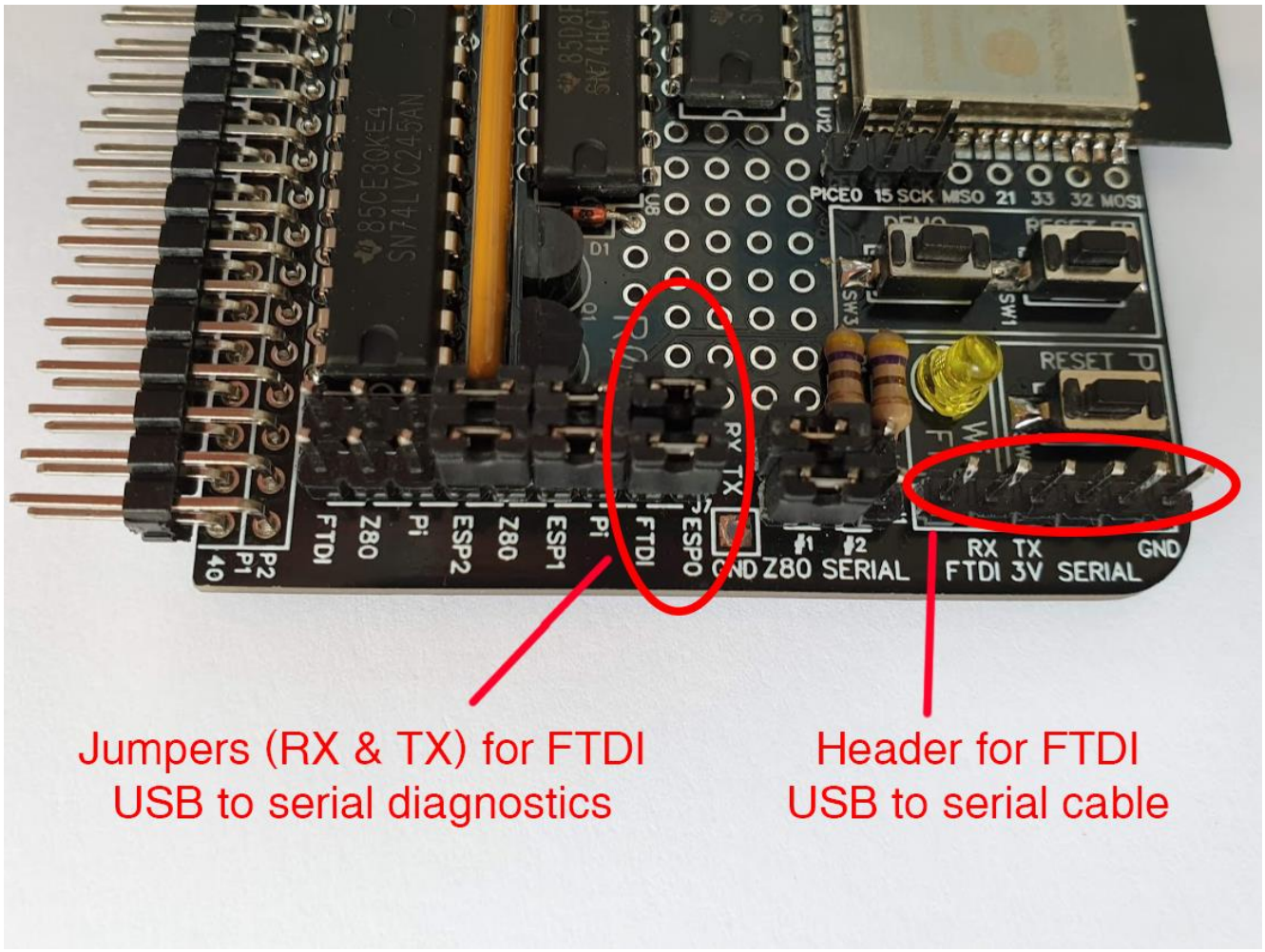
Connect the adapter to the pins shown making sure RX goes to the FTDI RX pin, TX to the FTDI TX pin and GND to the FTDI ground. The other pins are not used. Please note that a 3V cable must be used. There is a protection diode (D1) which may save the ESP32 if you inadvertently put 5V onto the RX or TX pins but I cannot guarantee it. If you are in any doubt about whether you really have a 3V3 cable, make sure the protection diode D1 is in place and connected the correct way around.

The connections for a genuine FTDI cable (black USB connector body) are:

- Black or Blue = GND (J1 Pin 1)
- Orange = FTDI TXD (J1 Pin 4)
- Yellow = FTDI RXD (J1 Pin 5)

And for the "Prolific Debug Cable" referenced above (blue USB connector body):

- Black = GND (J1 Pin 1)
- Green = FTDI TXD (J1 Pin 4)
- White = FTDI RXD (J1 Pin 5)



In addition two jumpers need to be in place to connect the FTDI Rx/Tx signals to the ESP32. These are on J5 (mid-way down the right side of the BusRaider PCB). You should put jumpers in place vertically to connect between Pi and ERSP1 for both the RX and TX columns as shown in the photo above.

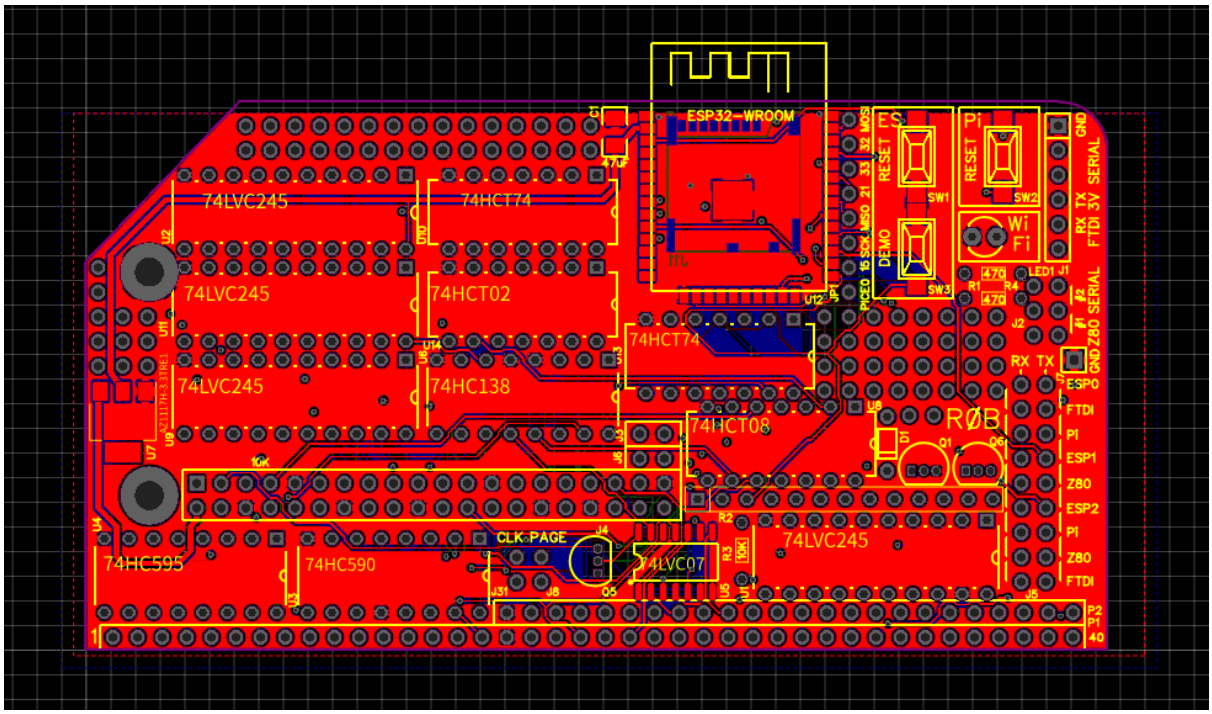
Once the connections and jumpers are in place run terminal emulation software such as TeraTerm with baud rate and settings 115200, N, 8, 1.

If this is the first time you have connected the serial port it would be advisable to use a terminal emulator such as TeraTerm (https://ttssh2.osdn.jp/index.html.en) or the Serial Monitor built into the PlatformIO VSCode add-in. The serial port settings are 115200, 8, N, 1 (but the baud rate 115200 is the only thing that generally needs setting).

With the board powered up you should see a regular (once every 10 seconds) message from the ESP32 on the serial port indicating its status including WiFi connection, etc. If you don't see this (or other similar activity) then check the jumpers and the FTDI cable connections.

APPENDIX 10 CONSTRUCTING THE BUSRAIDER

There is a construction video for BusRaider 1.7 https://www.youtube.com/watch?v=akkhFuqRgis and a series of videos for BusRaider 1.6 which are a little more detailed https://www.youtube.com/watch?v=CyE8oFtVzxQ



Please note that R3 is shown on the PCB as 10K but I have found that a 1K resistor works better in this position.

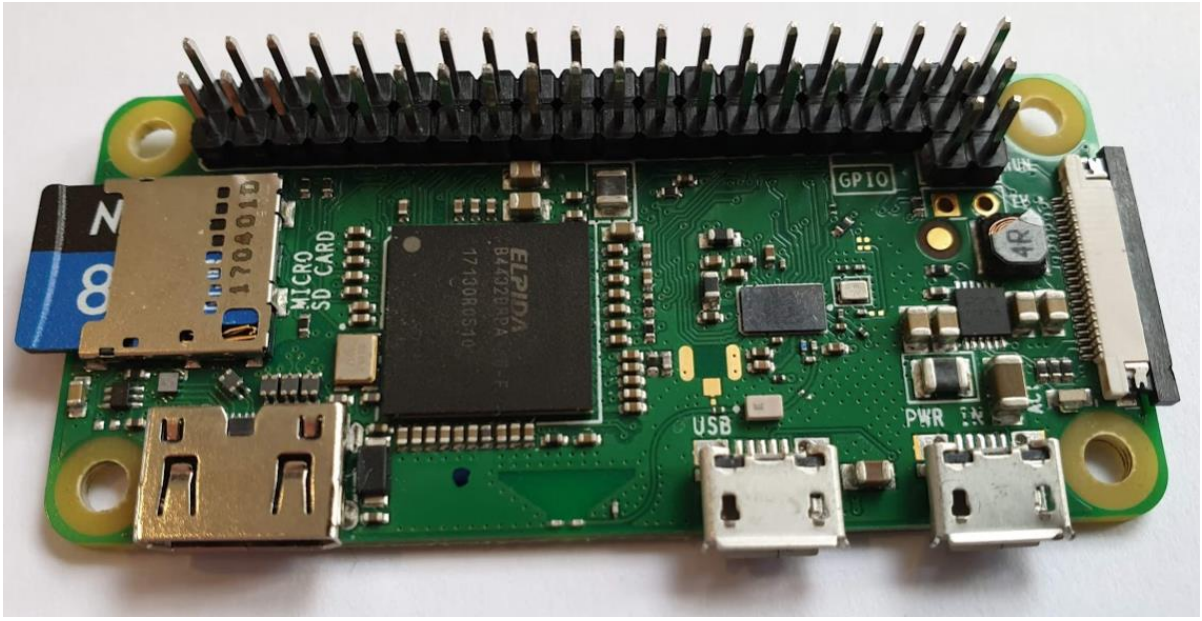
The key things to note when constructing are that the surface mount components should generally be attached first as they are lower to the PCB and sometimes awkward to add later. I generally find it easier to add components in height order with the least-high components being added first. This enables you to make use of gravity to hold components in place and avoids having to solder over a high component when trying to reach a lower one.

The use of sockets for the ICs is optional if you want to go down that path. I tend to prefer not to use them as they add an additional potential point of failure and there is little benefit unless you intend to experiment widely with different chip versions or similar.

Appendix 10.1 Pi Zero Installation

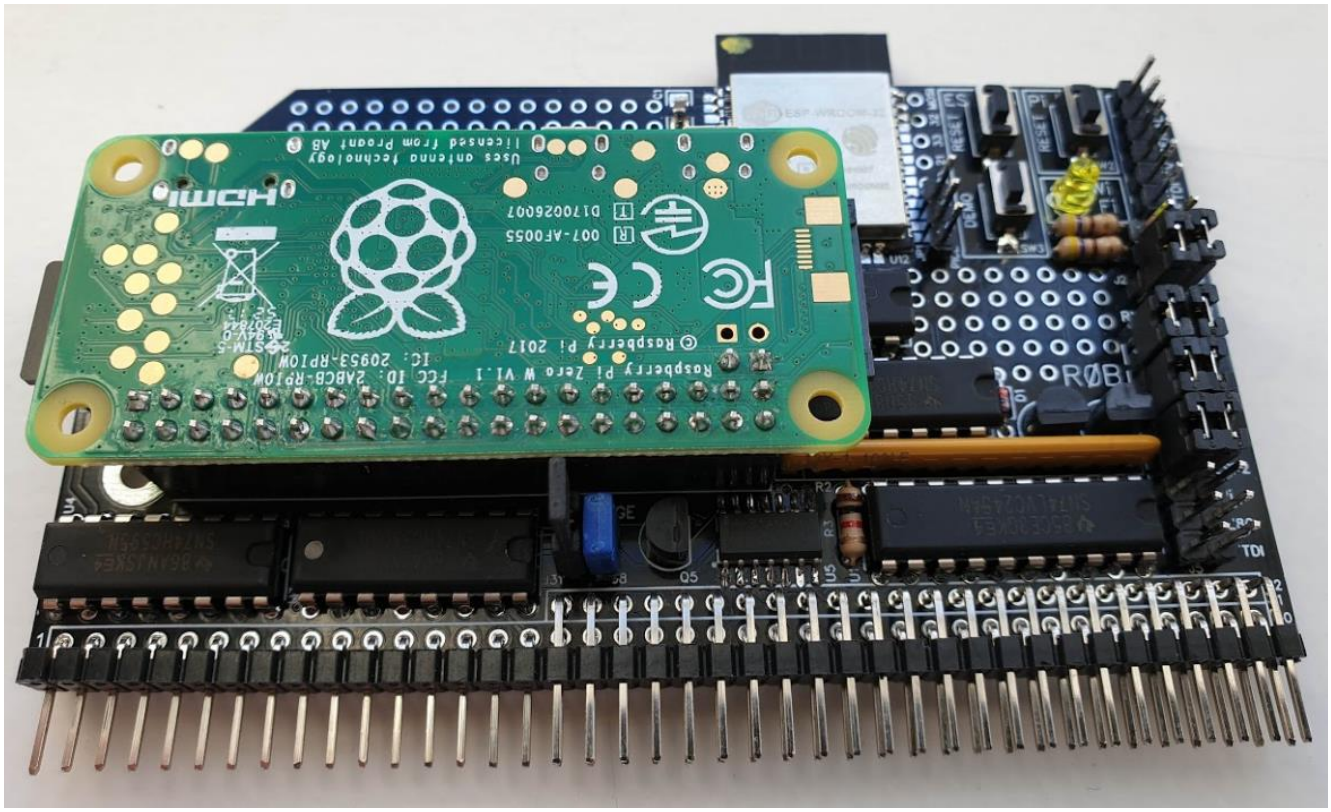
The Pi Zero (W) must have header pins (two rows of 20 or a single 2x20 header) installed in the “normal” way which means that the header is on the Pi Zero (W) component side. There should also be a 1x2 header in the holes marked

Run on the Pi Zero (W) – this allows the Pi Zero to be reset using the RESET button on the BusRaider. It is optional whether a header is included in the holes marked TV (this functionality is not used). See below:



Orientation of the Pi is such that the component side of the two boards face each other and the holes in the BusRaider on the left side (above and below the voltage regulator) line-up with the holes in the Pi Zero.

Finally, when it comes time to install the Raspberry Pi Zero (W) I have tended just to plug the Pi into its socket (J4 – just above the 74HC590 towards the bottom of the board) without additional supports. But if you want you could use one or two 10mm stand-offs with a washer (the gap is around 11mm) to secure the Pi better.



APPENDIX 11 INSPECTION AND TESTING

Once the board is fully populated it is time for inspection. I recommend using a magnifying glass or similar to inspect all soldered joints carefully looking for bridged tracks, dry-joints or similar. An awful lot of time and heartache can be saved by this simple step and getting into the habit of routinely inspecting your work before powering on is a really good thing to do. Take particular care with the power regulator area (because errors in this area can damage lower voltage components like the Pi and ESP32) and any area where soldered joints are very close together such as the 74LVC07 (U5), ESP32 (U12) and SD card socket.

Following a thorough inspection, you are read to test the board. I would initially recommend leaving the Raspberry Pi Zero (W) removed from its connector and initially concentrate on the ESP32 functionality.

If you have a USB to serial adapter cable (3V only) then connect this to the FTDI Serial Port connector (J1) as described in Appendix 9 and run your terminal emulation software (also described in Appendix 9).

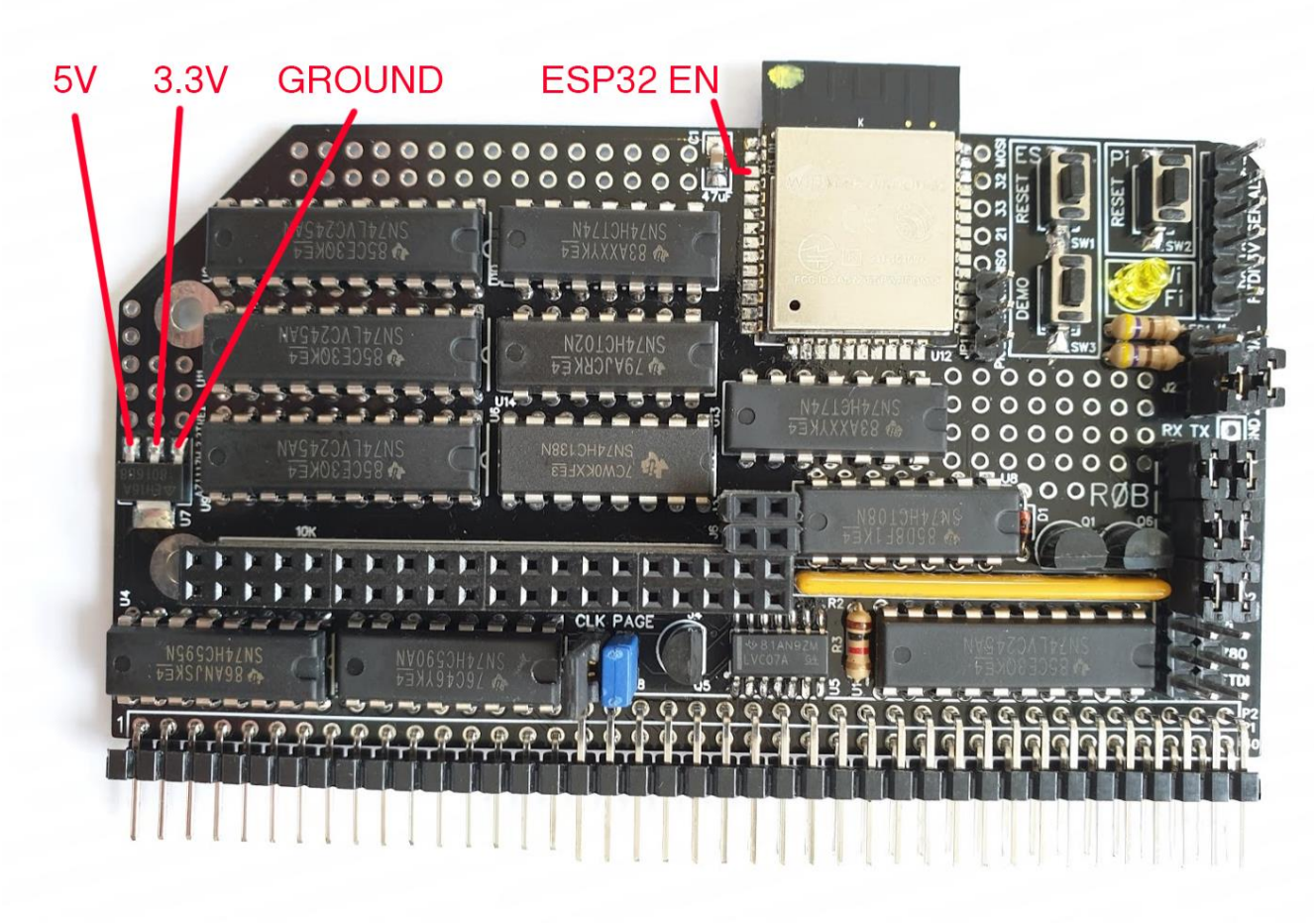
It is now time to power up the board. Insert it into the RC2014 backplane (making sure it is the right way around (Pin 1 is marked with a small 1) and then turn on the power. Assuming you don't see a flash of blue light and small acrid smoke (only kidding) then the BusRaider should be powered up.

If all is well, you should see something like the following appear on the terminal emulator:

```
rst:0x10 (RTCWDT_RTC_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0018,len:4
load:0x3fff001c,len:928
ho 0 tail 12 room 4
load:0x40078000,len:9280
load:0x40080400,len:5860
entry 0x40080698
ÿN: BusRaiderESP32 1.7.030 (built Feb 27 2019 16:57:45)
T: StatusIndicator: setup name wifiLed configStr {"hwPin":"13","onMs":200,"shortOffMs":200,"longOffMs":750}
N: StatusIndicator: wifiLed pin 13(13) onLevel 1 onMs 200 shortMs 200 longMs 750
N: FileManager: setup
{"spiffsEnabled":1,"spiffsFormatIfCorrupt":1,"sdEnabled":1,"sdMOSI":"23","sdMISO":"19","sdCLK":"18","sdCS":"4"}
N: FileManager: setup SPIFFS partition size total 1374476, used 109436
E (1260) sdmmc_common: sdmmc_init_ocr: send_op_cond (1) returned 0x107
W: FileManager: setup failed to init SD (error ESP_ERR_TIMEOUT)
T: ConfigNVS: Config wifi ...
T: ConfigNVS: Config wifi read: len(61) {"WiFiSSID":" ","WiFiPW":" ","WiFiHostname":"BusRaiderESP32"} maxlen 100
T: ConfigNVS: Config mqtt ...
[E][Preferences.cpp:38] begin(): nvs_open failed: NOT_FOUND
onfigNVS: Config mqtt read: len(2) {} maxlen 200
T: ConfigNVS: Config netLog ...
[E][Preferences.cpp:38] begin(): nvs_open failed: NOT_FOUND
NVS: Config netLog read: len(2) {} maxlen 200
T: WiFiManager: Event SYSTEM_EVENT_WIFI_READY
T: WiFiManager: Event SYSTEM_EVENT_STA_START
. . . . .
```

If you don't see anything at all on the terminal emulator and you do have some basic electronics tools – like a multimeter – then check that you see 5V appear on the regulator (which is located on the far left of the PCB when looking at the component side) – place the meter's ground probe (black) on the ground pin and positive (red) probe on the 5V pin when in DC Voltage mode.

If that's ok, then check between Ground and 3.3V to make sure you see this voltage – which is used to power the ESP32 and some of the logic chips.



If all is well with the power then check the solder joints around the ESP32 again and check that your FTDI cable is correctly connected. You can start looking at the voltages on other pins too – such as on the ESP32 Enable/Reset pin which is marked ESP32 EN in the image above – in this case the voltage should be around

APPENDIX 12 ESP32 DIAGNOSTIC MESSAGES

During normal operation the ESP32 shows a regular diagnostic message when an FTDI cable is connected and ESP32 to FTDI jumpers are in place. This message is as follows:

```
N: 90027 BusRaiderESP32 V1.7.030 SSID rdint01 IP 192.168.86.192 Heap 226132 Avg 64.75uS Max 17846uS Min 50uS
Slowest Command 17542, LoopTimer 3009
```

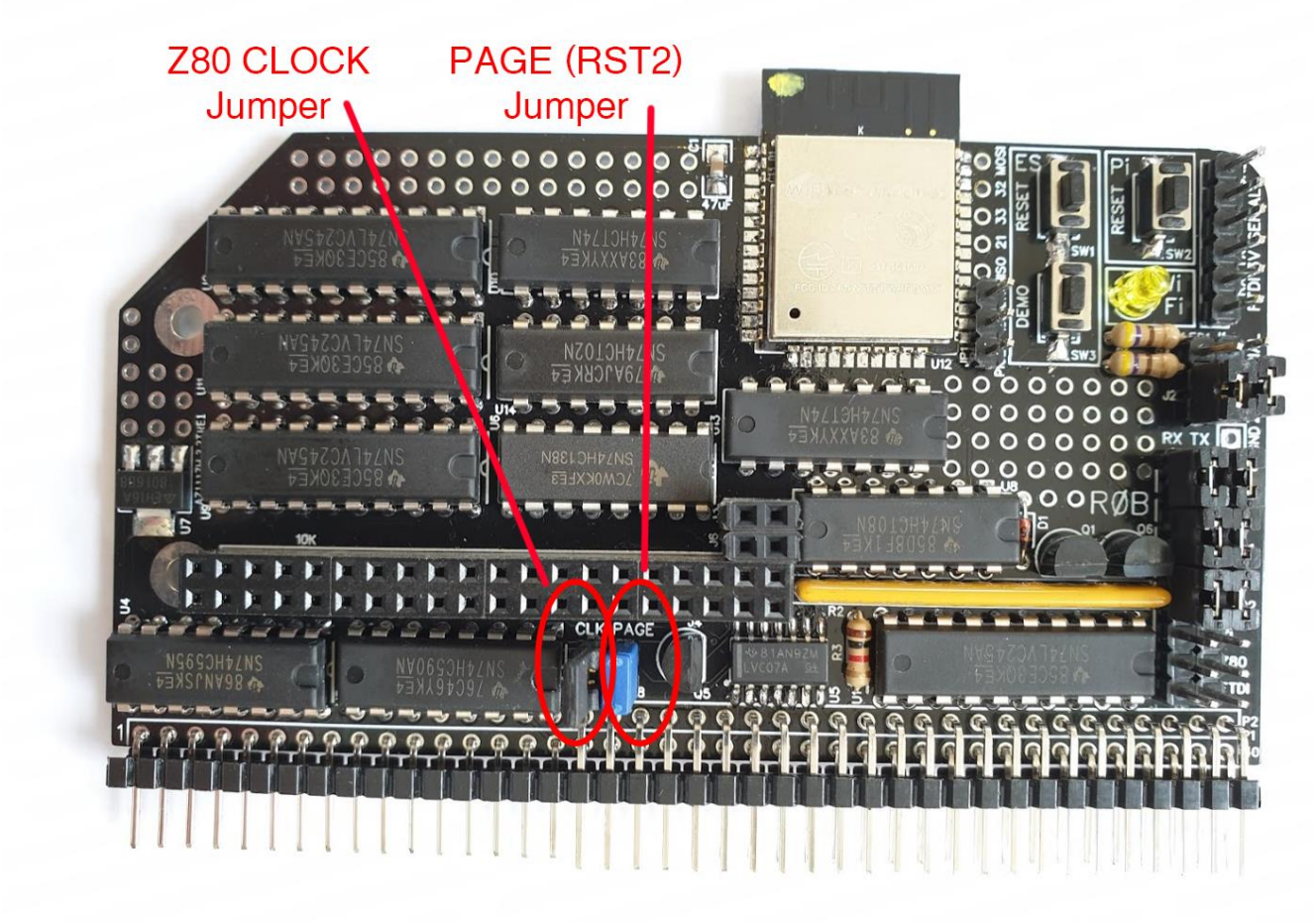
APPENDIX 13 PAGING AND THE PAGE JUMPER

Paging is an important aspect of opcode injection and, hence, single-step debugging. The principle is that when the BusRaider is injecting opcodes (and other data) onto the RC2014 bus there should be no conflict with other memory or IO subsystems that are also trying to place data on the bus. In order to support this functionality, the base memory system of the RC2014 must be disabled (or paged-out) when the BusRaider wants to inject opcodes and data onto the bus. This can be done using a PAGE line and there is some support for this with Spencer's 64K RAM card and the RST2 bus line on the PRO bus.

I have made a modification to my 64K RAM card to page the upper 32K of RAM in the same way that the lower 32K paging currently works. I did this by hacking the existing circuit and using part of an unused 74HC32 to disable the upper RAM block when the PAGE (RST2) line is active (low). I also installed a 1K resistor on the card to act as a bus

pull-up on the PAGE (RST2) line. I could provide more details if anyone was interested but, ideally, a memory card might be designed which had this functionality from the start.

In order for the PAGE signal to appear on the bus the PAGE (RST2) jumper needs to be installed as shown in the photo below.



APPENDIX 14 TARGET CLOCK GENERATION AND THE CLK JUMPER

The Raspberry Pi Zero (W) has the ability to generate an arbitrary clock frequency (with some limitations) in the range (approximately) 200KHz to 8MHz (actually it might be able to go higher than this but I haven't tried). The generation process is controlled using the data from the machine descriptor table – Appendix 7.1.2 – and attempts to emulate the speed of the real retro computer being emulated.

In order to enable the CLOCK signal onto the RC2014 bus the CLK jumper needs to be installed as shown in the photo above.

The clock speed can be adjusted in the WebUI.