TRS8BIT

PRODUCED AT WWW.TRS-80.ORG.UK

HI EVERYONE AND A WARM WELCOME TO THE MARCH 2023 EDITION, WHICH STARTS OUR 17TH YEAR!

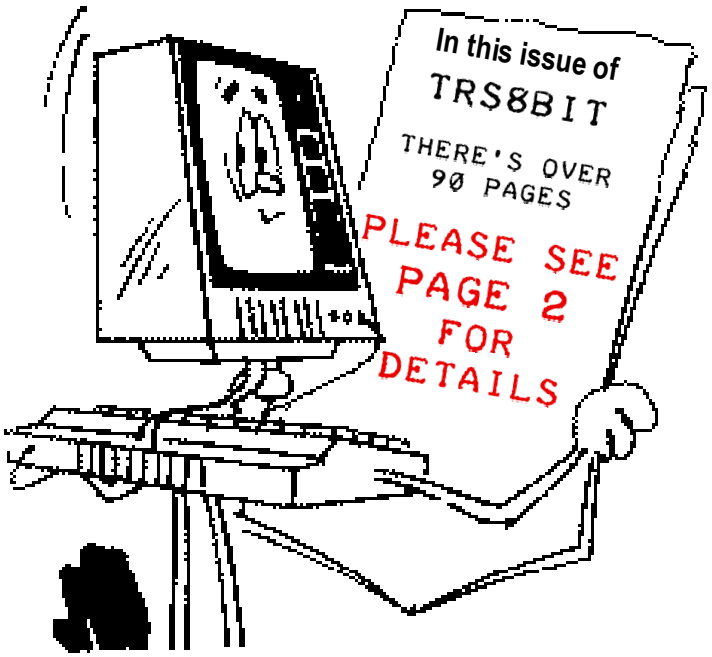
FIRST, LET ME THANK EVERYONE WHO HAVE CONTRIBUTED ARTICLES TO THIS BUMPER EDITION! THANK YOU. WITHOUT YOUR SUPPORT AND INTEREST THIS NEWSLETTER JUST WOULDN'T HAPPEN!

SECONDLY, GREAT NEWS FOR ALL TANDY FANS....

Tandy Assembly 2023

IS HAPPENING IN
SPRINGFIELD, OHIO
29SEP23 - 01OCT23
HAVE YOU BOOKED YOUR TICKET?

NOW, HERE'S A BIT OF SAD NEWS. I'M SORRY TO REPORT THAT MAV IS UNWELL AND RECOVERING FROM A SERIOUS ILLNESS SO THAT'S WHY THIS ISSUE LACKS AN ARTICLE FROM HIM. A STELLAR RUN OF ARTICLES, AT LEAST ONE PER ISSUE SINCE SEPTEMBER 2011, PERHAPS NOT EQUALLED BY ANY OTHER CONTRIBUTOR, WILL RETURN IN JUNE. I'M SURE WE ALL WISH MAV A SPEEDY RECOVERY AND LOOK FORWARD TO HEARING FROM HIM IN JUNE.



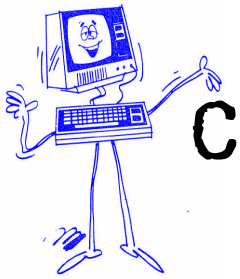
GEORGE AND PETER PHILLIPS ARE RELEASING THE LATEST UPGRADE OF THEIR TRS80GP EMULATOR. VERSION 2.4.11 AVAILABLE FROM THEIR WEBSITE, SHOULD BE UP AND RUNNING BY THE TIME YOU READ THIS.

SO, DOWN TO THIS ISSUE AND AS I'M SURE YOU CAN SEE, IS YET ANOTHER BUMPER ISSUE. IT STARTS OFF WITH AN ARTICLE ABOUT RUNNING THE TRS-80 ON THE 'NABU' AND, UNBELIEVABLY JUST GOES ON GETTING BETTER!. RUN A MODEL 1 FROM A USB PORT, STRING SCANNING, RE ENGINEERED HARDWARE, A GENIE III RELIVES, A NEW SERIES, 'HANDY HINTS' (JUST FOR FUN). DID YOU REALISE THE MASSIVE DIFFERENCE BETWEEN THE UK'S AND USA'S M100? WELL, THAT'S SORTED! PASCAL HOLDRY IS IN NEED OF THE ORIGINAL CASSETTE SOFTWARE FOR THE TC-8. PLEASE CAN YOU HELP?

AND FINALLY, THERE'S A FULL REPORT OF THE 2022 COMPETITION AND DETAILS OF THE 2023 COMPETITION, WITH, ONCE AGAIN, A FABULOUS PRIZE.

ENJOY THE READ!
SEE YOU ALL IN JUNE, UNTIL THEN -

TAKE CARE - DUSTY



CONTENTS

NEW VERSION 2.4.11

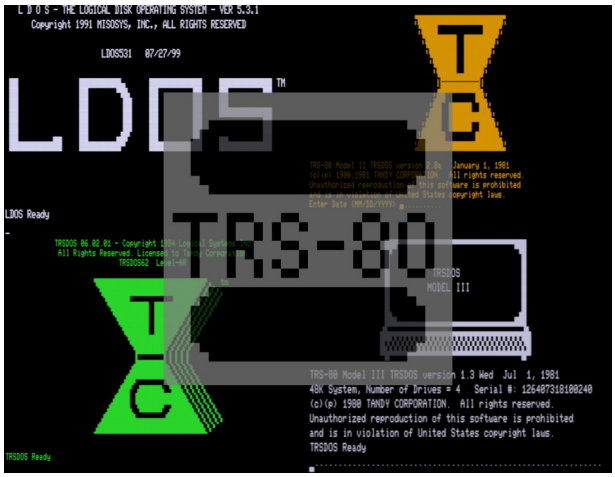
DON'T HAVE A TRS-80?

CRAZY PRICES ON EBAY?

TIRED OF RIFA CAPS BLOWING?

OF FLOPPIES SHEDDING OXIDE?

OR JUST PLAIN DEAD CHIPS?



THEN WHAT YOU NEED IS TRS80GP.
THE ULTIMATE TRS-80 EMULATOR

RUNS ON WINDOWS, MACOS, LINUX & RASPBERRY PI.
EMULATES MODELS 1, 2, 3, 4, 12, 16, 6000,
MC-10, DT-1, VIDEOTEX AND MICRO CO-CO.

JUST LIKE A REAL TRS-80 WITH NONE OF THE HASSLE. BUILT-IN DEBUGGER AND OTHER FEATURES GREAT FOR PROGRAMMERS.

GET IT NOW AT HTTP://48K.CA/TRS80GP.HTML

NEW VERSION 2.4.11, AVAILABLE NOW, FEATURING :-

PRELIMINARY DEVICE VIEWER.
SOUND EFFECTS WITH STEREO SOUND ON MACOS AND LINUX
DISK VIEWER SHOWS SECTORS IN ID ORDER INSTEAD OF POSITION

Model 1 L2 on the NABU

Ted Fried

The NABU Personal Computer has been getting a lot of attention recently thanks to the video by Adrian Black which re-kindled interest in this vintage computer and pointed out that someone was selling about a thousand of them which were new in the box, so I purchased one!

Unfortunately there isn't much one can easily do with this machine, however there are a few projects underway which port CP/M to it and also creating a system which can serve over the network the original games that this computer ran.

I thought I would make a contribution by implementing something totally different which took advantage of the NABU's motherboard components...

I thought it would be an entertaining project to use my MCLZ8 to emulate the Zilog Z80, the TRS-80 Model I ROM BIOS and DRAM, and remap the keyboard and video to the motherboard resources of the NABU Personal Computer. Now the NABU is effectively acting like a Tandy TRS-80 Model 1!

This project was built quickly thanks to already having most pieces from existing projects and only took about an afternoon to complete.

The first step was to confirm that the MCLZ8 would work in the NABU as a replacement CPU for the Z80. The MCLZ8 was previously tested to work on the TRS-80 Model III which has a slower bus speed than the NABU, but it worked the first time without an issue.

Next I added some storage arrays in the MCLZ8 to store all of the NABU's configuration IO accesses to the video chip and keyboard. This saved me a lot of time so I would not need to learn how these chips worked from the datasheets!

I then added the TRS-80 BIOS ROM and RAM emulation into the MCLZ8 so that all accesses to these memory ranges were handled in the Teensy using an array in C.

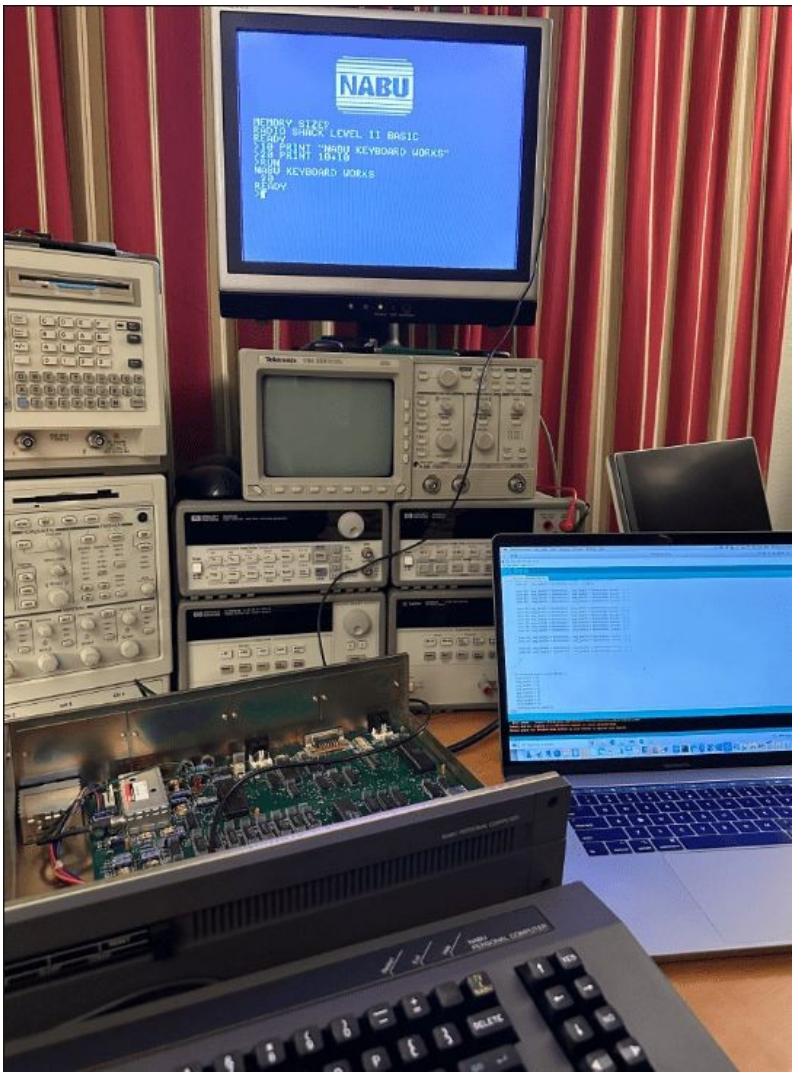
The NABU and TRS-80 have different video sizes with the NABU being 40×24 and 64×16 for the TRS-80. Since the NABU had more vertical space I was able to start the TRS-80 video downwards 8 lines which left room to keep the NABU splash screen which I thought is a neat effect! The TRS-80 has more columns so the NABU can only display 40 of them. The characters are still there in video memory, they are just not displayed. I used an array to convert the TRS-80 row/column mapping to the NABU which is faster than performing the math…

The NABU keyboard sends keystrokes as ASCII characters to an Intel 8251A on the motherboard, so I only needed to poll the UART's status bit and read characters when they arrive. The TRS-80 performs parallel keyboard polling instead of using a UART so I needed to convert from the NABU's ASCII to the address bit map for each TRS-80 keyboard polling address. This work was done on one of my previous projects which I was able to reuse.

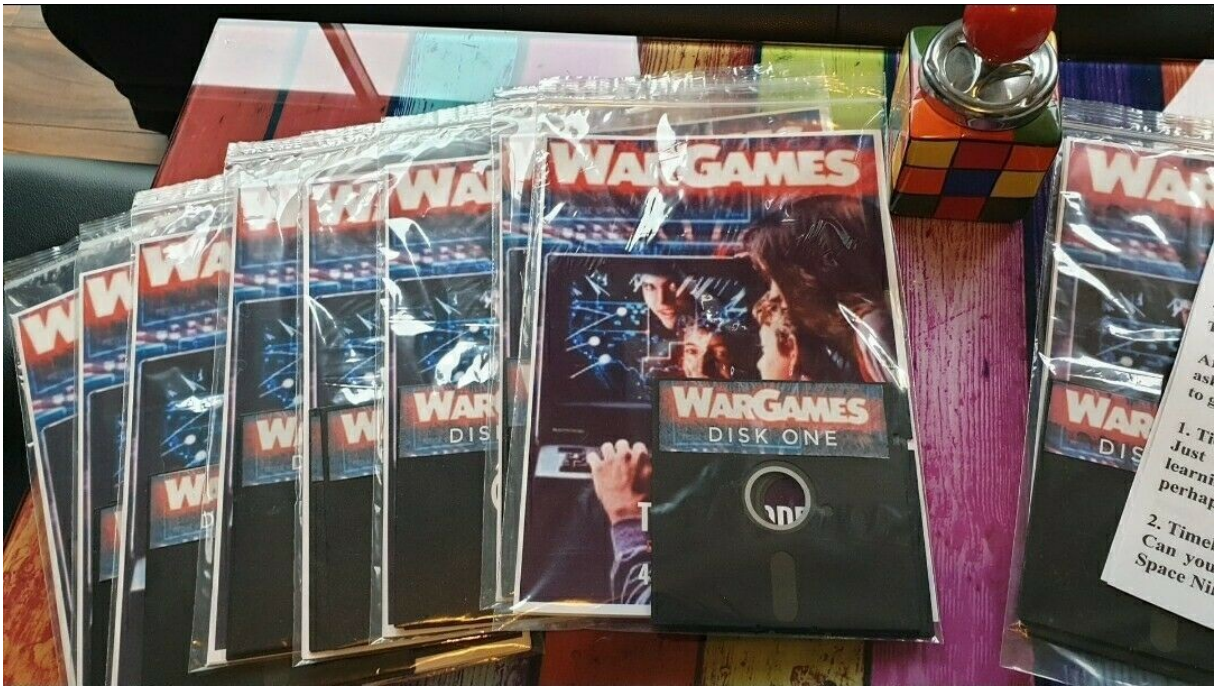
So, that's about it! The NABU now boots to the splash screen and then runs the TRS-80 Model I Level II BASIC. Below are a few pictures of the machine in action.

All of the source code is uploaded to GitHub.

Ted Fried


NABU
MEMORY SIZE?
25610 BYTES FREE
NABU SHACK LEVEL II BASIC
READY
>10 PRINT "NABU KEYBOARD WORKS"
>20 PRINT 10-10
>RUN
NABU KEYBOARD WORKS
 20
READY
>
DELETE


NABU
SHARP



All new from Dreamland Software.

This two disk package comprises what is in effect the FULL Dreamland Software collection.

Based on the classic 80's film 'Wargames', your entire experience is controlled by the W.O.P.R computer from the aforementioned movie.

Greetings Professor Falken.

New titles including (of course) Global Thermonuclear War. (Ouch.)
Retro Football Manager. (Based on the ORIGINAL 80's classic, a full soccer management simulation.)
The Battle of the Mutara Nebula. (If you are a Star Trek fan, then you know FULL WELL what this is.)
And the Kobayashi Maru Scenario. (Another one for the Trekkers.)

Also included, are **ALL** games released by Dreamland Software since inception four years ago.

The hugely successful Actual Reality. It's sequel Actual Reality II - Battlestar Raven And RockStar.

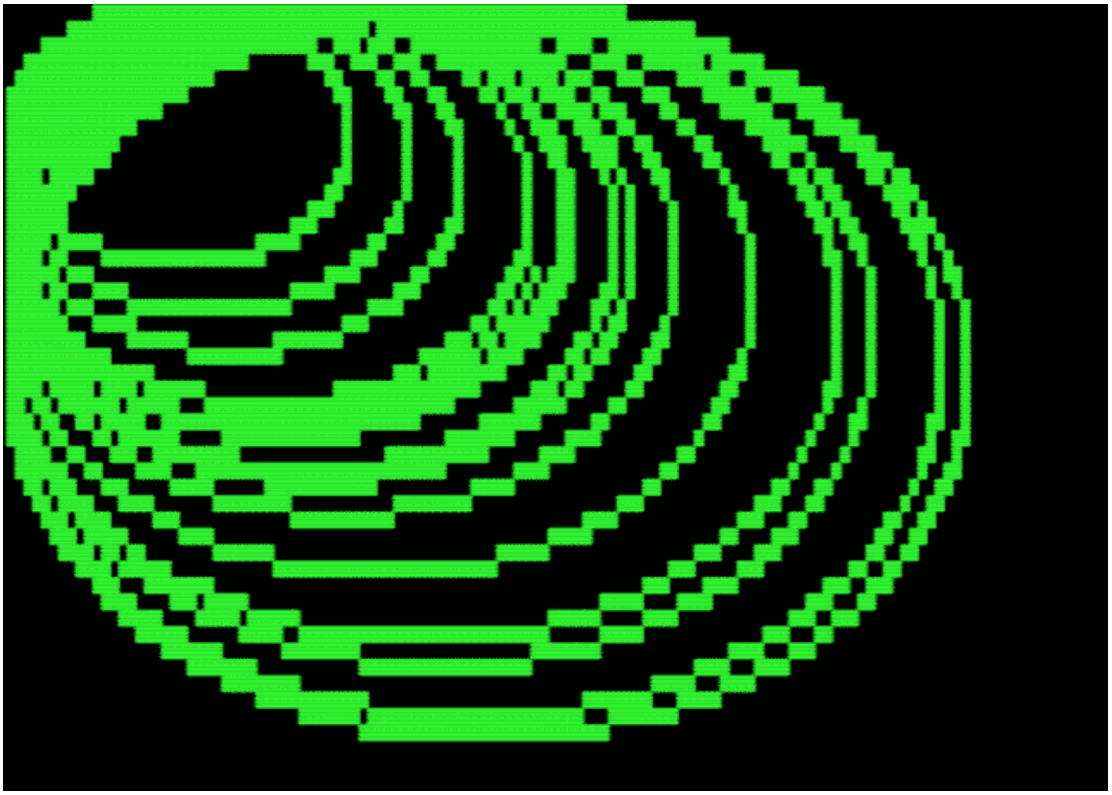
14 Programs in total (Including the W.O.P.R Master Program.)
Both disks full to capacity whilst remaining bootable.

Full video preview available here:
https://youtu.be/XXD1IG8-CO0

Please note that this item is currently available for Model One ONLY.
Model Three/Four will be available soon. (Watch for update.)

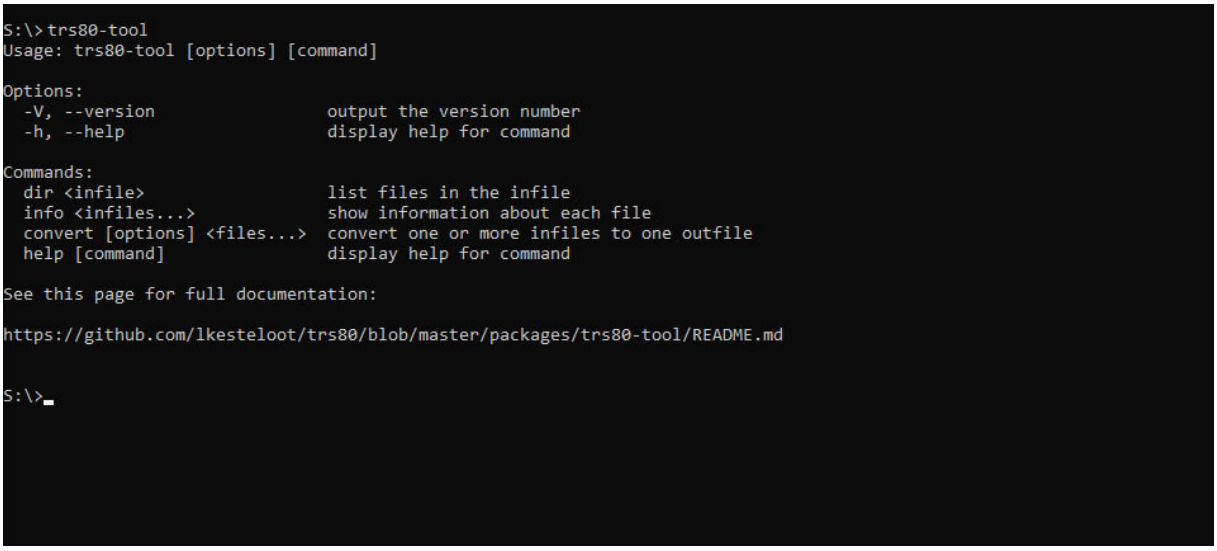
E.T. FONEHUME

HAPPY NEW YEAR DUSTY. I WAS PLAYING ON MY TANDY TO MAKE A ONE-LINER CIRCLE CREATION PROGRAM AND WAS CARRIED AWAY ON A 'NEW YEAR' EXCITEMENT 'BURST'! HERE'S A 'TIME TUNNEL' IMAGE CREATOR WHICH IS BEST RUN ON, AS ALWAYS, TRS80GP BUT WITH THE TURBO OPTION SWITCHED ON, UNLESS, OF COURSE, TIME'S NOT A PROBLEM.



```
0 RANDOM:CLS:FORT=0TO1STEP0:D=RND(90)+30:FORX=.02TO6.5STEP2/D:SE
T(((SIN(X)+1)*.5*D),((COS(X)+1)*.2*D)):NEXT:NEXTT
```

Lawrence Kesteloot has rewritten, his TRS-80 Cassette command line tool to support many new functions. Called "TRS80-TOOL" it is a command-line tool for manipulating TRS-80 files.



Lawrence Kesteloot has updated his "TRS80-TOOL" to v2.3.1

It runs under NODEJ, Linux, macOS, and Windows. It can be downloaded from Lawrence's site or locally from the The Virtual Tape Utilities page on this site.

The new version adds support for multi-file CAS files (such as CLOAD Magazine or Temple of Apshai) and will try to guess (and set, if you want) the starting address of a SYSTEM file.

The main page for TRS80-TOOL can be found at

https://github.com/lkesteloot/trs80/tree/master/packages/trs80-tool

along with instructions.

It runs under NODEJ, Linux, macOS, and Windows. It can be downloaded from Lawrence's site, the The Virtual Tape Utilities page on this site or right here for Windows / Linux / Mac.

TRS-80 Model 1 USB-A Power Mod (5V DC conversion)

Tim Halloran

This modification can be useful if your Radio Shack power brick is bad or noticeably hums. You might not have one at all. The modification lets your Model 1 run significantly cooler inside. I've tested this modification on two Model 1 keyboards (one with the Rosser 64K in-the-keyboard modification select *Upgrading the Model I to 64K of DRAMs (pdf)* from the menu that the link brings up on Ira's site) and run them for multiple days.

I strongly suggest you do not do this modification on your only working Model 1 unless you are quite skilled with electronics work. The modification is pretty simple but does require one trace cut and the removal of several components from the motherboard. Several wires are then soldered in place. Finally, a 5-pin DIN to USB-A cable is constructed.

This image shows the resulting system.

Key features of this modification are

The outside of the computer is unmodified and is indistinguishable from a stock Model 1. As you will see below, I use a sticker and orange paint on the 5-pin DIN socket to distinguish my prototype units so I use the right power cable.

It won't work, but is safe, to connect a

real Radio Shack model 1 power brick. It is very easy to make this mistake. I have several times.
Power still comes in via the 5-pin DIN connector.
The power switch still works (so does the reset).

MEAN WELL power supply is recommended

The modification creates the ability to use USB power, but I recommend you setup a MEAN WELL RS-15-5 AC to DC power supply. This has very low ripple and noise (on par with the original power brick). It also works on AC power worldwide. One of these can power a keyboard and an expansion interface in my tests.

My experience leads to this rule of thumb. If you changed the memory and have a 5V only keyboard a USB charger tends to work fine as power. If you want to use an expansion interface, or you kept the 4116 memory (using a DC-to-DC converter) then you need to use the MEAN WELL.

Two modification options

There are two options, one I consider GOOD and the other BAD:

(OPTION GOOD) This replaces the 4116 memory with either 4517 (16K) or 6665 (64K used as 16K). This simplifies the modification and allows the entire computer to run off 5V DC power. I've had far less trouble with this modification.

(OPTION BAD) keeps the 4116 memory in the computer but requires a small DC-to-DC converter. I've found this is less successful, and more trouble. It seems to only work with the MEAN WELL linear power supply (where you have clean power and can dial in the voltage to about 5.1V. This also tends to bring out weaknesses in the video sync logic. Be prepared to swap out Z5, Z6, and Z57 with new chips.

A word of caution

I strongly suggest starting with a working Model 1. Do not do this modification unless you are comfortable working inside a Model 1. If you have never opened one up odds are you will destroy your computer. I can almost guarantee that the flimsy stock keyboard connector will be damaged. Consider removing the original keyboard connector and installing headers or another more robust solution of your choice. You'll see my "quick connect" approach in the pictures.

This modification is harder than installing a lowercase modification but easier than doing the Rosser 64K in-the-keyboard modification.

What you need

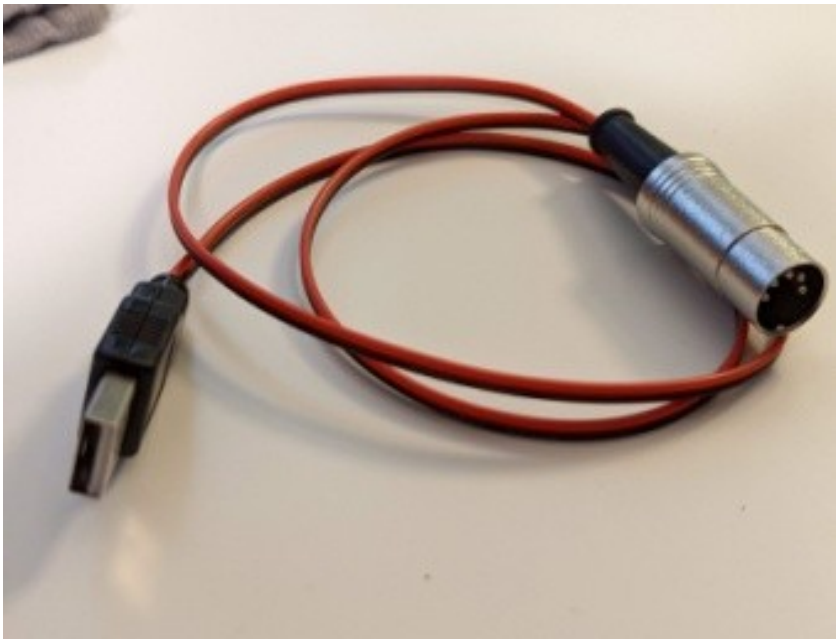
Here's the list of parts you'll need

- One male 5-pin DIN solder connector
- One male USB-A solder connector
- Wire at least 22 gauge (to carry DC power). Silicone is recommended. For the cable I use parallel red-black wire. Inside the computer it doesn't matter. Do not use wire-wrap gauge wire for any part of this modification!
- **(OPTION GOOD)** Eight MCM4517P15 (16K) or MCM6665BP20 (64K used as 16K) memory chips. Search AliExpress or eBay for deals (or reach out to sellers like Mav). Memory is socketed (thank goodness) so this is easy to install. Ensure what you buy is in a 16-pin DIP package. Also if you upgraded a CoCo 2 from 16K to 64K, the 16K memory you took out is 4517 memory (and will work). Further, if you have done the Rosser 64K in-the-keyboard you are all set. Rosser's modification already converted your memory to 6665 chips which only require 5V.

(OPTION GOOD) A 14-pin DIP socket and a 14-pin DIP 7 position switch. I'll show this below, however, you can just use a short wire if you wish.
(OPTION BAD) A DC to DC converter 12V/-12V. I used this one, pick +-12V no pins.
What tools do you need
• A good desoldering tool that you are comfortable using. I use a Hakko 301.
• A good soldering iron. I use an (older) Weller WES51.
Tools to take the Model 1 apart. I assume you can take apart a Model 1. (If not please see YouTube as there are lots of videos showing this.)

Building the power cable

Below is a picture of the cable you need to construct. I might have one, please reach out to me on Facebook or Discord and I'll try to send you a cable. Constructing the cable is a difficult task to accomplish (DIN connectors are a pain to solder).

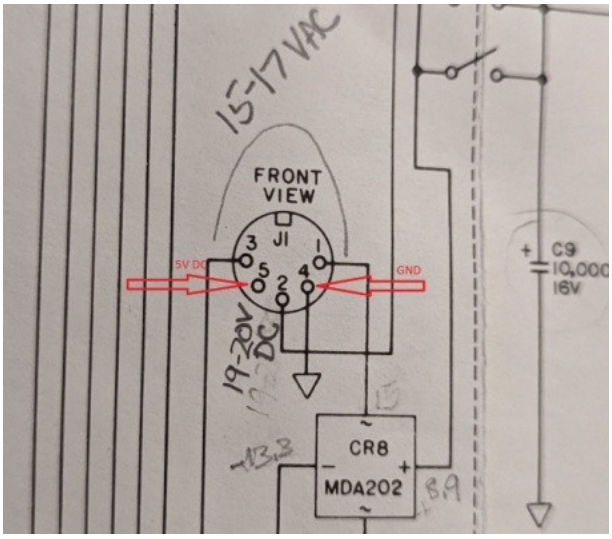


Caution

Do not cut and "reuse" a USB-A cable to save time. This is a bad idea. We want to use 22 gauge wire (or larger) to ensure good current flow. The Model 1 keyboard uses ~0.9 amps. A lot of commercial cables are intended for data and use very thin wire. Soldering a USB-A plug is easy, it's soldering the DIN that is difficult.

Some background and theory behind our cable

To understand this cable, we need to look at what we are doing and why the Model 1 is somewhat complex to convert to DC power. The image below shows the pinout of the Model 1 power connector from the *TRS-80 micro computer technical reference handbook*.



This image explains why it is not easy to just create a replacement power brick and plug it in. AC and DC power is sent into the keyboard. Pins 1 and 3 send in 15 -17V AC (I measured this at 15V into the bridge rectifier CR8 on my machine). ~20V DC comes in on pin 2 with Ground on pin 4. So 15V AC and 20V DC come into the computer.

What the computer actually needs to operate (inside) is three DC voltages: 5V, -5V, and 12V. However only the 4116 memory uses all three voltages. The remainder of the motherboard only uses 5V.

This modification keeps ground unchanged on pin 4. We'll send 5V DC in on the unused pin 5.

Solder the 5-pin DIN first

Take apart the DIN and solder the black wire to pin 4 and the red wire to pin 5. Place a small piece of heat shrink on the wire to slide down after soldering. You should get something like the next picture.

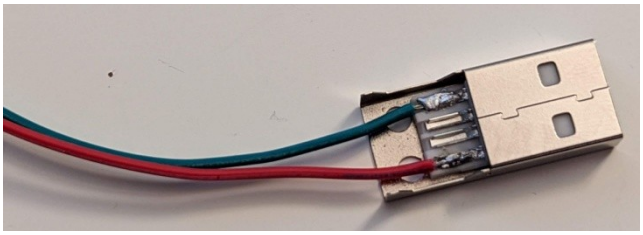
One suggestion is to tape the DIN (pins down) to the edge of the table and bring the wire down while soldering to the back. I do not recommend that you clamp a pin to an alligator clip holder and solder, I've found that this approach often melts the plastic holding the pins. Resulting in a bent pin.

Ensure you give the connections a good tug. If they pull apart you did a bad solder job and need to try again.

Next put the entire DIN connector together. We started with this connector because you have to slide the cover over the wire. This is easier before the USB-A connector is attached.

Solder the USB-A connector second

Ensure the entire 5-pin DIN is soldered and put back together. Half the cable is done. The male USB-A connector gets soldered as shown in the image below.



Note in the image above red is 5V and green is ground. This is a connector I keep around to check which side is which. Your cable will probably use black for ground.

Ensure you give the connections a good tug. If they pull apart you did a bad solder job and need to try again.

You then snap the USB plug casing around the metal. I tend to use

some CA glue and clamp the two sides together to ensure the casing doesn't come apart over time. This glue is not required, but I suggest you keep it in mind if plug casing keeps coming apart as you use it.

Your cable is complete. It is a good idea to test it with the continuity tester on your multimeter.

Modifying the Model 1 computer

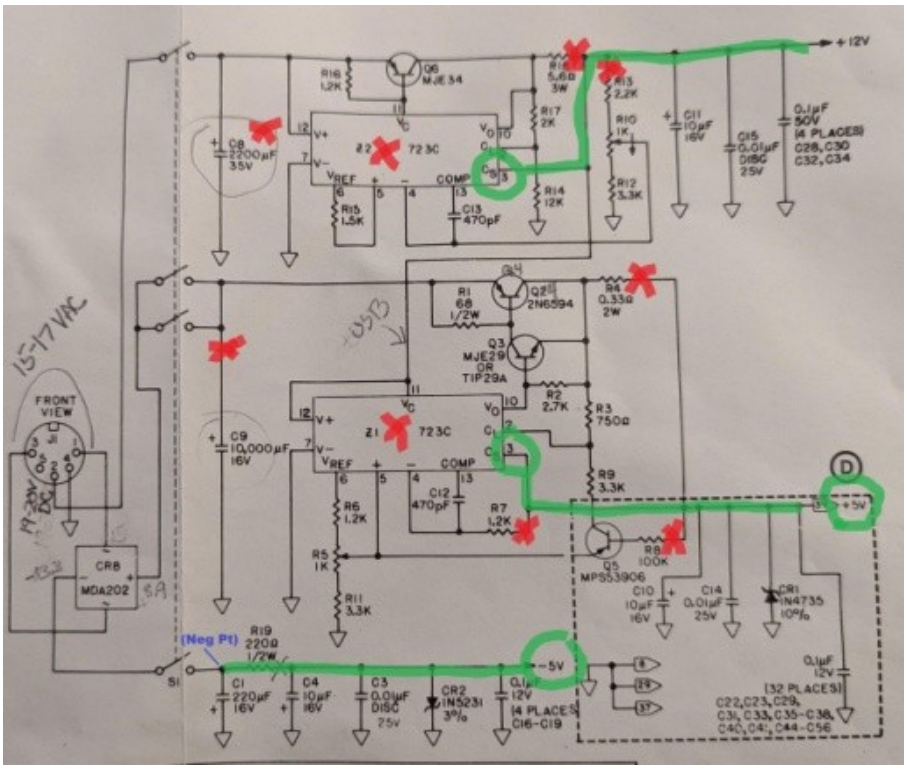
In this section I'll describe how to modify your Model 1 motherboard. We'll start by explaining what we are doing and then go into the step-by-step.

Some background and theory behind our changes

Changing a Model 1 to use DC (even batteries) has been done in the past. One instance that I'm aware of is discussed in TRS-80 Trash Talk Episode 18. Mike Yetsko created a battery powered TRS80. In this episode Mike describes using batteries to supply all three voltages 5V, -5V, and 12V. If you really know the computer anyone can do this. It is more complex to keep the modification simple. My first prototype was a tangle of wires and cuts. Its gotten better.

What we want to do is bypass the power regulation circuits on the motherboard. The trick is to channel the power around the active power regulation circuits (that create the heat in a stock Model 1). Also we do not want to remove the big transistors (this was required in the CoCo but is not in the Model 1). What we want is a clean path to the DC power out to the voltage rails. My pencil marks give some

insight into earlier attempts and can be ignored. The red-X's show where we will remove components and the green show the clear path out of the power regulation circuits to the motherboard logic.



(OPTION BAD) KEEP 4116 MEMORY: To use the original 4116 memory we have to supply each power path with the voltage that it needs. However, there is a subtle shortcut that we can do for the -5V. DC-to-DC transformers are cheap, however we will get 12V and -12V not 12V and -5V. What can we do? The bottom path of the schematic above is a circuit to regulate -13.3V (my measurement yours will vary) out of the bridge rectifier (CR8) to -5V. What we will do is connect our -12V DC to the *(Neg Pt)* (the negative or right side of C1) to use part of the Model 1's power regulation to get us -5V. This works very well in practice and does not generate excessive heat because the regulation here is not active, its just a diode (CR2). In fact, CR2 is a 5% 5.1V Zener diode. Perfect.

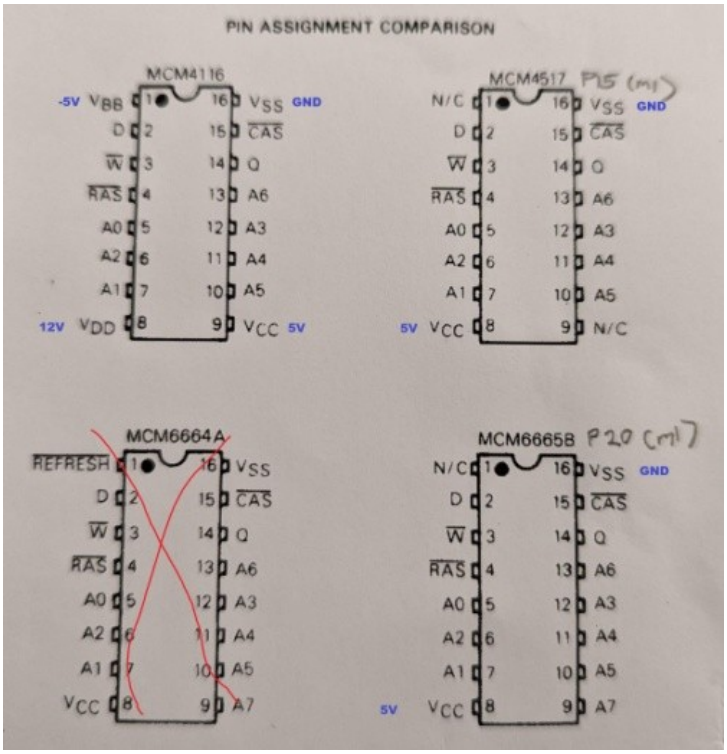
Given the DC-to-DC converter the rest is easy. 5V is hooked up to Z1 pin 3 and 12V is hooked up to Z2 pin 3. This makes all the power rails exactly like the original machine.

To summarize this option we

- Connect 5V DC power into the 5V power grid (the left hand side of R4 as presented below) and into the input of the DC-to-DC converter.
- Connect -12V out of the DC-to-DC converter to *(Neg Pt)* to get -5V. Connect 12V to Z2 pin 3.

The drawback of this approach is we have to include the DC-to-DC converter. I've also noted it brings out problems in the video sync circuit. The benefit is that we do not need to replace the memory chips.

(OPTION GOOD) USE 4517 OR 6665 MEMORY: If we swap out the memory we can make a 5V only system. This removes the need for the DC-to-DC converter, however, there is snag. If you look at the memory chip pinouts for the 4517 (16K) and the 6665 (64K using 16K) in the diagram below, in both cases, we have to get 5V on the lines where previously there was 12V. We can safely ignore the -5V lines.



To get 5V where the stock computer had 12V is easy. Simply connect Z1 pin 3 to Z1 pin 12 (connected to Z2 pin 3). This will channel 5V over to the (old) 12V lines and make them 5V as well.

To summarize this option we
• Change all the memory chips (Z13 through Z20) from 4116 to either 4517 (16K) and the 6665 (64K using 16K).
• Ignore the -5V path (bottom of the diagram above) it will not be used.
• Connect 5V DC power into the 5V power grid (the left hand side of R4 as presented below). Bridge Z1 pin 3 to Z1 pin 12 (connected to Z2 pin 3) to change the 12V path to 5V.

Note: If you already did the Rosser 64K in-the-keyboard modification some of the above is not need. That modification disconnected the 12V lines and bridged them to 5V. So this change will be easier as we discuss below.

Remove the DRAM chips

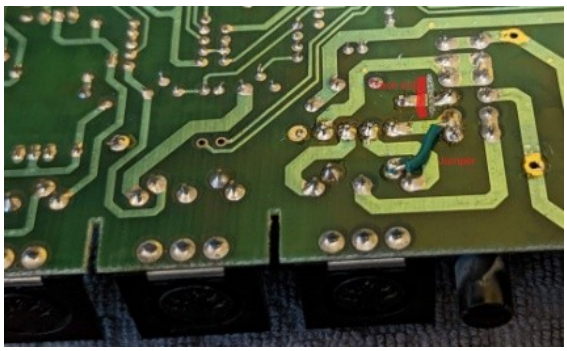
Start off by removing the socketed Z13-Z20 memory chips.
Regardless of option we want to test power before we put memory chips back into the machine.

Removing components

We'll start by removing several components. Required for both options.
• Remove two DIP chips: Z1 and Z2
• Remove resistors: R4, R7, R8, R13, and R18
• Remove capacitor C8 entirely.
Unsolder and disconnect only the + side of C9 (near the power switch).
Remove things carefully and save them. Worst comes to worst you can reverse the modification. Below is a picture of what this should look like.

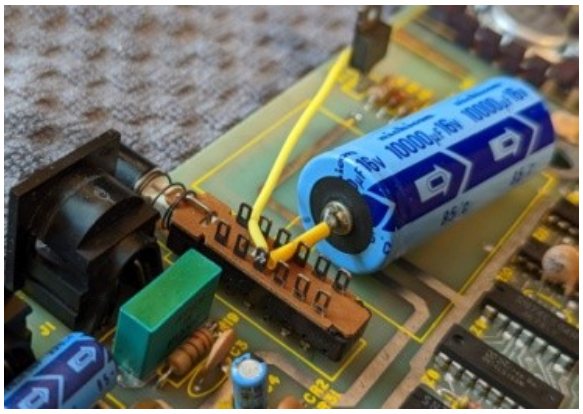
Setup the power switch

Flip the computer over to the back and focus near the power connector and power switch. We need to do a trace cut and solder a wire from pin 5 of the power connector to the right position on the switch. None of this is labelled so use the image below to guide you.



Use the continuity tester on your multimeter to ensure this is all correct. Check that the connection on pin 4 of the power connector is not shorted to the pin close to it. Also be sure your trace cut is not conducting. I used a Dremel tool to do the trace cut (just to be sure) but great care is required (practice this a few times if you try it).

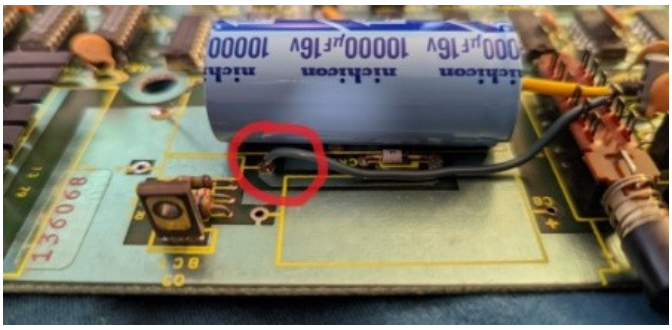
The trace cut isolates the power switch lead we will use to connect power to the motherboard. To do this turn the board around to show the chips. You need to position the positive lead of C9 to reach the third connection back on the left hand side of the power switch. Add a bit of shrink wrap to ensure the C9 lead doesn't connect with any other power switch terminals. We'll also insert a wire of about 4 inches that will connect to the motherboard. The picture below shows what this should look like prior to soldering.



I suggest you look over the leads on the switch. I've had to use some sandpaper to clean some of these off. If it looks black or dirty do this. We want a good solder connection. Then solder the two wires to the power switch.

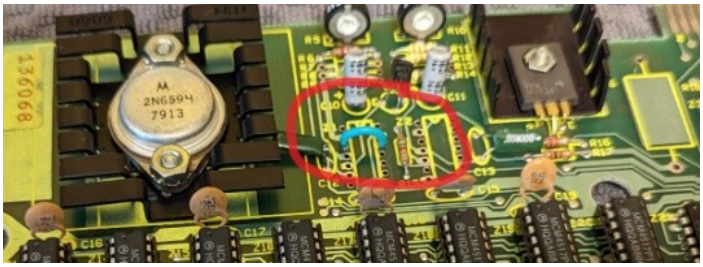
At this point the procedure differs depending on which option you choose.

Completing (OPTION GOOD) Using 4517 or 6665 Memory

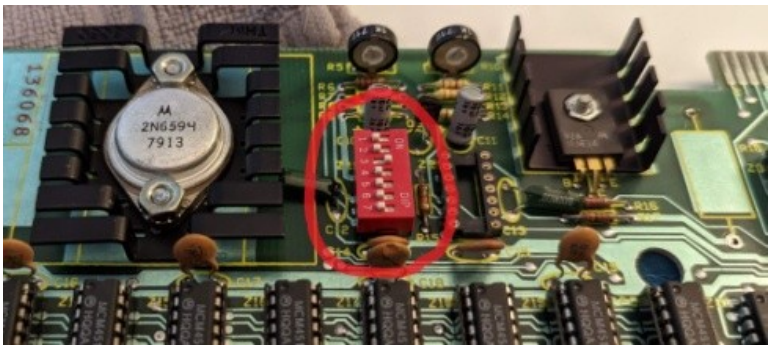


Trim and solder the wire connected to the power switch to the R4 hole closest to the power switch. This should look like the image below.
Note: Do not do the next step if you have Rosser 64K in-the-keyboard modification. The modification already did what the next step asks you to accomplish. If you have Rosser's modification then you are done.

Finally, we need to bridge the old 12V grid to the old 5V grid so everything is 5V DC. You can do this one of two ways. The first option is to simply put a short wire between Z1 pin 3 and Z1 pin 12 (straight across). This is shown in the image below.

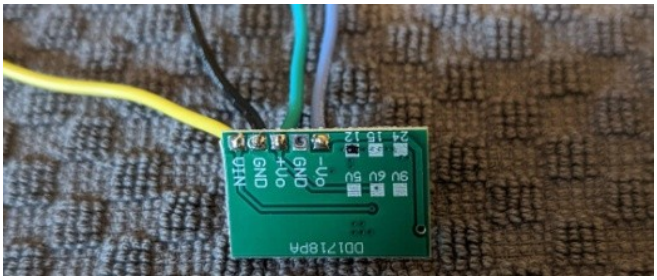


The second option is to solder sockets into Z1 (and optionally Z2) and use a 7 switch DIP. This is shown in the image below. Be sure to turn off all the switches except 3 (which is on).



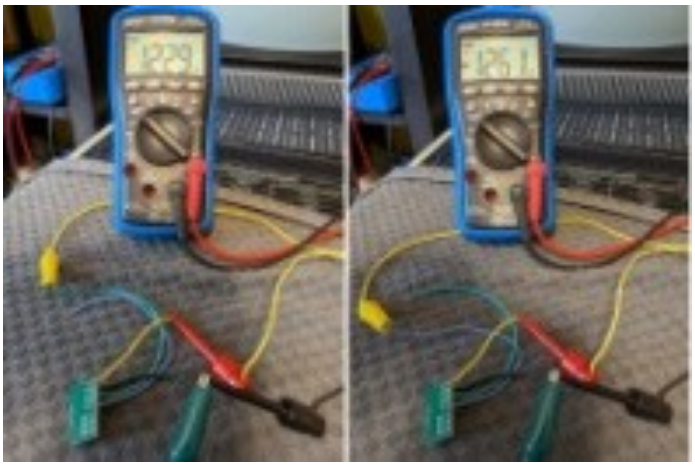
It really doesn't matter which you choose (I used the second when I was testing). The modification is complete (you can skip the next section).

Completing (OPTION BAD) Keep 4116 Memory

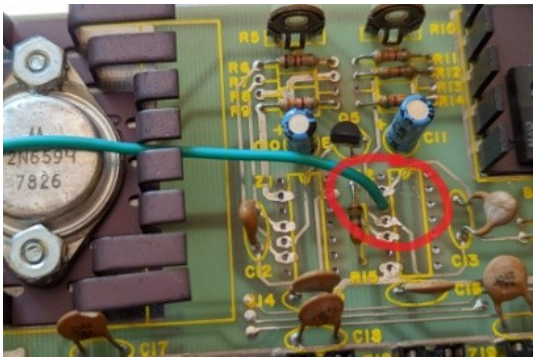


First we need to wire up the [+-12V DC-to-DC converter]. Check that ground in and out are common (be sure!).

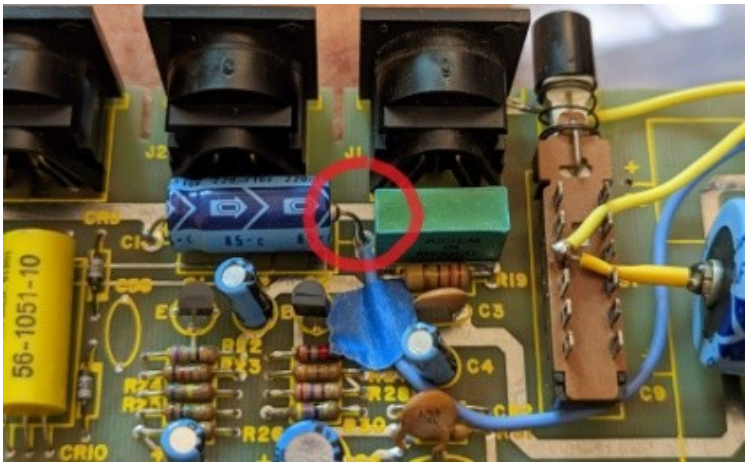
The output wires need to be longer than the VIN and GND wires. I strongly suggest you test this converter before installing (with a bench power supply or a custom USB cable). I've had these be mislabelled and give me, for example, +-9V rather than +-12V. You should get voltages like those shown in this image.



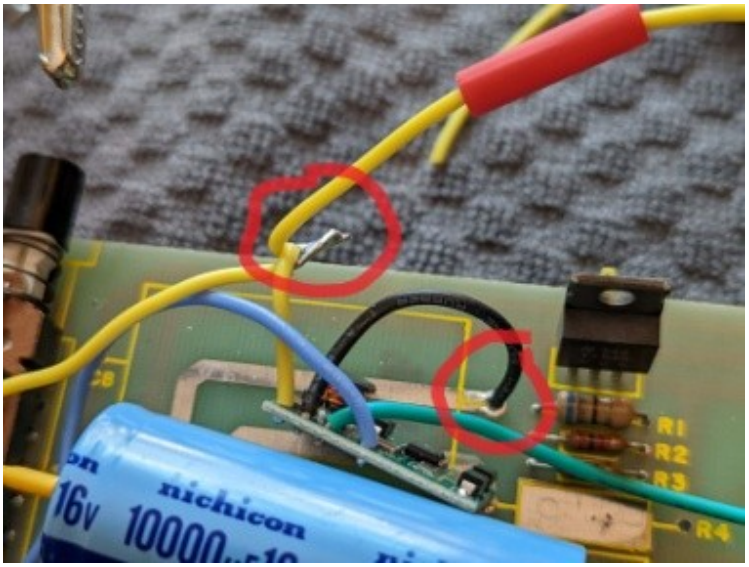
First solder the 12V wire from the converter to Z2 pin 3 as shown in the image below.



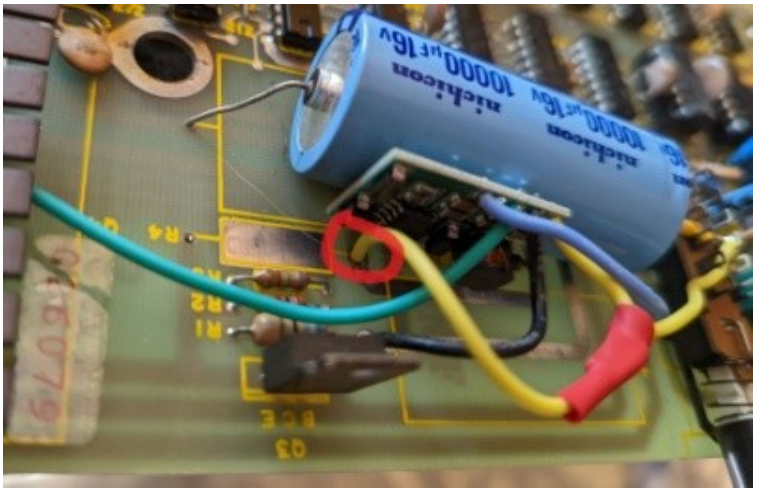
Next solder the -12V wire to the side of C1 closest to the power switch as shown in the image below. This is tricky to do. Tape down the wire so it stays in place.



Next solder the GND wire from the converter to the hole for C8 farthest from the power switch (see the image below). We also need to connect the wire from the power switch with the input wire from the converter to a new third wire. The third wire will connect to the motherboard. Use some shrink cover on this connection.



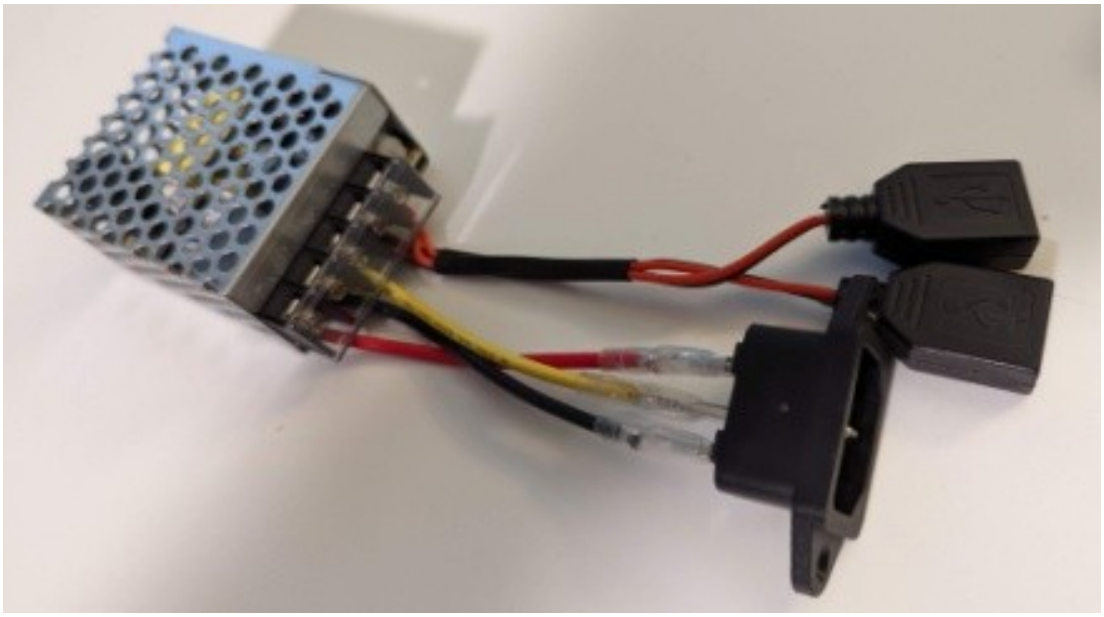
Trim and solder the new wire you just added (and soldered into a bundle) to the R4 hole closest to the power switch. This should look like the image below.



The modification is complete.

Setting up a MEAN WELL power supply

Before we test I want to show how to use a MEAN WELL RS-15-5 AC to DC power supply. This has very low ripple and is recommended. This is simple to setup. But you need a plug adapter and a female USB -A connector. My prototype looks like the image below.



You can create one as you wish. A few notes

- Mine uses USB-A female connectors because I want to be able to quick connect to either USB-A power or the MEAN WELL. This seems to be nice to have. But you could just connect the GND and 5V DC to the DIN power cable if you wish.

- I have two USB-A female connectors because I use this to power both a Model 1 keyboard unit as well as a prototype Expansion Interface (EI). This has the advantage that both have a common GND (helps stability). I'm not yet done testing the EI setup (but it is quite similar to OPTION BAD).
- This could clearly be packaged better but I haven't figured out how to do so.

If I haven't stressed this enough already. This is highly recommended, especially if you choose OPTION BAD.

I suggest adjusting the MEAN WELL output voltage to about 5.1V.

Testing your modification and putting the memory back

Ensure your power switch is off on the Model 1 computer. Hook up the power cable and connect it to a USB charger or a MEAN WELL. For all this testing I suggest you use a clip to connect the negative lead of your multimeter to the side of C9 not connected to the power switch (as we do on a stock Model 1).

(TEST 1: Any Power?) Before powering this on check the pin on the power switch one up toward where you press the switch. This should read around 5V DC.

(TEST 2: Motherboard Grid Okay?) Power on the machine. For OPTION GOOD test that pin 8 of any of the DRAM sockets (any of Z13 through Z20) is 5V DC. For OPTION BAD we need to test several pins of a socket (any of Z13 through Z20). Pin 1 should be -5V DC. Pin 8 should be 12V DC. Pin 9 should be 5V DC.

If any of these tests fails. Stop, something is wrong. Double-check the instructions and see if you shorted something with a solder connection.

If these tests pass shut off the computer and disconnect the power cable. Carefully insert your memory. On the bench connect the power cable, you monitor cable, and the keyboard. Turn on the monitor and then the computer. It should work as a Model 1 is expected to work. If you computer doesn't go. Some suggestions are

- Check voltages when the machine is turned on. Do they look okay? Troubleshoot to get power correct.
- If you did OPTION BAD and the screen is scrolling? Try adjusting the MEAN WELL voltage a bit. If this doesn't help you likely need to swap out Z5, Z6, and Z57 with new chips (as I noted up above). If you are using USB power (not a MEAN WELL) you may never get OPTION BAD to work. Last resort, convert to OPTION GOOD.

Use the *TRS-80 micro computer technical reference handbook* to isolate the problem.

You can reverse the modification (if you were careful with Z1 and Z2). Just socket Z1 and Z2, undo all the new stuff, and put all the old parts where they belong. Move the wire under the power switch to reconnect the trace cut. I've done this several times. The only real danger is that you destroyed Z1 and Z2 when you took them out.

(Optional) Marking the power modification more obvious

I have plugged in modified Model 1 computers with the wrong power source. To avoid this, the solution I came up with is to paint the power socket orange and put a *5V DC* sticker below the power socket.

My approach is in the image below.

PASCAL HOLDRY, IN FRANCE, IS LOOKING FOR THE CASSETTE SOFTWARE TO GO WITH THE RE-ENGINEERED TC-8

CAN YOU HELP? PLEASE EMAIL ME WITH DETAILS AND I'LL PASS THEM ON TO PASCAL.

I'M TRYING TO LOCATE A COPY OF THE PROGRAM "ENHANCED BASIC". THIS WAS AVAILABLE, IN THE U.K. FROM HARDINGS (MOLYMERX). IT NOT THE 'EXTENDED BASIC' OR M'SOFT'S LEVEL III BUT MUCH SMALLER AND NEATER, GIVING ALL LEVEL 2 PROGRAMERS MANY OF THE DISK BASIC COMMANDS. I'VE PUT OUT REQUESTS IN VARIOUS SITES, BUT, SO FAR, TO NO AVAIL! (*SEE ADVERT ON PAGE 91 ED*)

AS ALWAYS, OF COURSE, I'M ON THE LOOKOUT FOR ANY SOFTWARE FOR THE U.K.'S **ACULAB 'FLOPPY TAPE'**. THERE WERE VERSIONS SPECIFICALLY 'ZAPPED', FOR USE ON THE AFT, OF ELECTRIC PENCIL, SCRIPSIT, ZEN, LEVEL III BASIC AND VISICALC. THERE WAS ALSO THE ENHANCED BASIC, XBAS. IT'S SO DISAPPOINTING TO

AT THE READY PROMPT

THERE ISN'T A 'MOD' RESERVED WORD IN LEVEL 2 BASIC, SO PETER SHOWS US 2 WAYS TO FIND THE REMAINDER OF A DIVISION.

A MOD M IS A-M*INT(A/M) FOR REALS, ALTHOUGH IT _MIGHT_ BE WRONG BECAUSE OF MATH ROUNDING IN SOME CASES.
IF YOUR NUMBERS ARE SMALL AND THE MOD IS A POWER OF 2 MINUS 1 (LIKE 7 OR 255) YOU CAN USE AND: A AND M

PETER PHILLIPS

```
READY
>LIST
10 A=34:M=8
20 PRINTA-M*INT(A/M);
30 PRINT A AND (M-1)
READY
>RUN
 2  2
READY
>_
```

UNKNOWN LOADING A SYSTEM TAPE
ML ARNAUTOV

LOADING A SYSTEM TAPE, OF UNKNOWN NAME, CAN BE A PROBLEM. HAVING LOOKED AT THE LEVEL 2 ROM, I WORKED OUT THIS SOLUTION.

TO LOAD A SYSTEM TAPE OF UNKNOWN NAME, ON A MODEL 1 LEVEL 2, OR A VIDEO GENIE, RUN THE FOLLOWING PROGRAM.

```
10 FOR I=16924 TO 16932:READ J:POKE I,J: NEXT: END
20 DATA 49,136,66,205,147,2,195,231,2
```

NOW PREPARE THE TAPE AS YOU WOULD FOR A NORMAL LOAD. TYPE 'SYSTEM' AND REPLY TO THE PROMPT *? WITH /16924 INSTEAD OF THE PROGRAM NAME. THEN SIT BACK AND WATCH YOUR PROGRAM LOAD.

THIS SIMPLE SOLUTION RELIES ON THE FACT THAT THE STANDARD LOAD PROCEDURE ALLOWS PROGRAM NAMES TO BE ABBREVIATED DOWN TO A SINGLE CHARACTER. WHILE THE NUMBER OF POSSIBLE 6 CHARACTER NAMES IS LARGE, THE NUMBER OF CHARACTERS WITH WHICH THEY START IS NOT!

EXTRACTING A CASSETTE FILE NAME
CLIVE DAVIDSON

HERE'S A SIMPLE SOLUTION TO FIND THE NAME OF A CASSETTE PROGRAM. THIS SMALL BASIC PROGRAM WILL REVEAL ALL.

```
5 CLS
10 INPUT -1,A$
20 REM EXTRACT THE FILE NAME
30 B$=MID$(A$,2,(LEN(A$)-3))
40 PRINT"FILE NAME IS";B$
```

NOTE THE FILE NAME IS ALWAYS PRECEDED BUY 'U' AND '<'. THIS INFORMATION IS REMOVED BY LINE 30, TO GIVE THE ACTUAL FILE NAME.


Joe Kay
Director
NORTH COMPUTER WEST MUSEUM
Create Experience Learn
joe@nwcomputermuseum.org.uk
01942 582826
Information Technology facilities & training
Workshops for programming, Electronics & VR
Community classes, clubs & retro games
IT Skills for the young & old
City & Guilds Creditation
Supporting education of local people
Creating opportunities & careers
Disable friendly
Volunteer-led training & museum operations
NORTH WEST COMPUTER MUSEUM
Create Experience Learn
www.nwcomputermuseum.org.uk
Leigh Spinners Mills, Park Lane, Leigh, WN7 2LB

String Scanning Routines
Vince Otten

In this article, I show how I developed a general set of string scanning routines which look for not just a single character, like INSTR(CH$, SR$), but any member of a character set (like whitespace, alphanumerics, "+-*/", etc.)

I needed to be able to scan a line like "program -option1 filename" and pull out the individual "tokens", storing them in an array named AV$(),--like C's argv[]--and keeping a count in AC%--like C's argc :

```
AC%         2
AV$(0)      program
AV$(1)      -option1
AV$(2)      filename
```

("program" has 2 arguments, "-option1" and "filename".) In the process of creating that, I developed a nice little general-purpose string-scanning routine which
1. tests for or against a character set (like all digits, or whitespace, or "() +-"; and
2. remembers where in the string it left off scanning.
Assuming the string to be scanned was stored in SR$, my program was able to

```
50200 GOSUB 9000
```

and the first token would be returned in TK$. The entire loop is:

```
50200 GOSUB 9000
50210       IF SC% THEN
          AC%= AC% + 1
          AV$(AC%)= TK$
50220       IF MR% THEN
          GOTO 50200
```

We get the next token (GOSUB 9000). But what if TK$ doesn't contain a token? The "GetToken()" routine at line 9000 also makes sure to only set SC% ("Success") if TK$ isn't empty. So only IF GetToken() is Successful do we increment AC% in line 50210 and fill the corresponding AV$() with the token (AC% is initialized to -1 in line 50060, not shown). GetToken() also keeps track of where it's scanning, ready to look for the next token. If there's still "more string to scan", MR% will be set, so in line 50220 we loop back to ask for another token.

How does GetToken() know where to begin when it first starts scanning SR$? Whenever we assign something new to SR$, we'll need to call InitScanString() in line 6000, which sets up a number of "scan state variables". Here's a typical application of that:

```
50099 '
      PROMPT FOR TEST STRING:
            REJECT ANY EMPTY ""

50100 PRINT: PRINT "ENTER STRING, IE 'COMMAND LINE (PROGRAM
ARGS... )'"
50110      LINE INPUT SR$
50120      IF SR$ = "" THEN GOTO 50100

50149 '
      SET UP SCANNING VARIABLES FROM NEW SR$

50150 GOSUB 6000

50199 '
      SCAN & RETRIEVE TOKENS LOOP:

50200 GOSUB 9000
50210     IF SC% THEN
                AC%= AC% + 1:
               AV$(AC%)= TK$
               ' SUCCESS: SAVE TOKEN IN AV$()
50220     IF MR% THEN
               GOTO 50200
               ' MORE TOKENS TO SCAN
```

My scanning routines don't do a lot of error-checking, so here I make sure SR$ isn't empty before I begin to use them. Then I call line 6000 to initialize the "Scan State Variables", which keep track of where we are currently scanning from in SR$, what the length of SR$ is, etc. We'll look at InitScanString() / 6000 later on.
Here's what GetToken() at line 9000 looks like.

```
7999 '
      2. NEXT TOKEN -- SKIP WS, RETURN TOKEN AFTER TOKEN
                        UNTIL MR% IS F
9000 '
9010 TK$= """' START WITH EMPTY TOKEN$
9020 SC%= 0' "RESULT" (IE, SUCCESS/FAILURE)
9100 GOSUB 7000' SKIP WS
9110 IF NOT MR% THEN GOTO 9500
9120   GOSUB 5500' "GATHER UNTIL" CS$ (WS)
```

```
9130   IF SC% THEN
           TK$= MID$( SR$, CC%, SL%)
9500 GOSUB 5900' UPDATE CC%= NC%
9990 RETURN
```

We start with a blank TK$ (line 9010), and assume we're "not successful" (9020). Then we call SkipWS() / line 7000 to move our search start point in SR$ (the "current cursor", CC%) over past any "whitespace" (blanks, carriage returns, linefeeds, stored in the scan's "character set", CS$). SkipWS() is kind enough to let us know whether there's something other than whitespace remaining in SR$ by setting the "more" flag, MR% (9110).

If there's more to scan, we GatherUntil() / 5500, ie, until we run into more whitespace. This routine reports the "selection length" SL% of the substring starting at CC% that does not match the whitespace in CS$, as well as providing a "succeed/fail" flag in SC%. Only if it's succeeded in identifying a non-whitespace token do we actually return that as TK$ (9130).

In any case, we UpdateCursor() / line 5900 to move the current cursor CC% to where the "new cursor" NC% is, just past the end of the token (9500).

SkipWS() is implemented in two steps: specify the character set to search for (space, linefeed, or carriage return); and SkipWhile() (line 5200) that set is matched.

```
6999 '
     1. SKIP WS

7000 CS$= " " + CHR$( 10) + CHR$( 13) ' WHITESPACE CHARACTERS
7050 GOSUB 5200' "SKIP WHILE" OR "IGNORE" WS
7090 RETURN
```

By now we've seen two examples of a set of four generalized scanning options:

SkipWhile()	line	5200	scan while matching a member of the character set
SkipUntil()		5300	scan until a member of the set is matched
GatherWhile()		5400	identify substring that matches the set
GatherUntil()		5500	identify substring until a member of the set is matched

"Skips" simply "move the cursor", whereas "Gathers" identify a span of 1 or more characters in SR$. Examples of how each could be used:

SkipWhile()
moving past all whitespace, as in our program here.
SkipUntil()
in Pascal, a comment begins with a { . We can skip comment contents by using SkipWhile() to look for the matching } .
GatherWhile()
Get the next variable name by setting CS$ to "alphanumerics" (a collection of all uppercase & lowercase letter + all digits)
GatherUntil()
Collecting a substring until whitespace, as in our program here.

SkipWhile() is coded :

```
5199 '
      "SKIP WHILE" ("IGNORE") --
            SKIP UNTIL NOT IN CS$, UPDATE CC%, MR%

5200 NF%= 0' "MATCH" CHAR SET
5210 GOSUB 5000' SCAN
5220 GOSUB 5900' UPDATE CC%
5230 GOSUB 6900' DETERMINE IF THERE'S "MORE" TO SCAN
5290 RETURN
```

NF% is the "Not Flag". If it's set (to -1), then the main GeneralScan() (5000) routine understands to not match what's in CS$; otherwise, we match at least 1 member of CS$, as in this demo. Both "Skips" will call UpdateCC() (5900) to "move" the current cursor CC% to where GeneralScan() has placed the new cursor NC% (even if NC% hasn't changed). Finally, we call IsThereMoreToScan() (6900) to signal whoever's called SkipWhile() that there may or may not be more information in SR$.

SkipUntil() isalmost exactly the same: it does set NF%, and then simply jumps to line 5210, above.

```
5299 '
      "SKIP UNTIL" ("FIND") --
            FIND 1ST MATCH W/ CS$, UPDATE CC%, MR%

5300 NF%= -1' "DON'T MATCH" CHAR SET
5310 GOTO 5210' SCAN; UPDATE CC%; DETERMINE IF "MORE" TO SCAN
```

Before addressing GeneralScan(), here's UpdateCC():

```
5899 '
      UPDATE CC%= NC%

5900 CC%= NC%: RETURN
```

Why not simply type CC%= NC% whenever we need to "update CC%"? It's not as though using GOSUB 5900 saves any memory—always a consideration on a TRS-80. But it saves having to memorize the steps, and it avoids misprints, like accidentally typing CC= NC%.
We also need to

```
6899 '
       DETERMINE IF THERE'S "MORE" TO SCAN

6900 MR%= CC% < LN%' IF WE'RE NOT AT END OF SR$, THEN "MORE"
6920 IF NOT MR% THEN
           MR%= NOT FN IC%( MID$( SR$, CC%, 1), CS$)
           ' IF CHAR AT CC% NOT IN CS$, THEN "MORE"
6990 RETURN
```

This is a bit more complicated. First, we check if the current cursor CC% has been moved all the way to the position of the last character in SR$, ie the "length" LN% of SR$. If it's not, then obviously we have more to scan, so MR% will be set to -1, "true". On the other hand, say SR$ was " A" (5 spaces and an 'A'). Once we'd skipped all the whitespace and updated CC%, it would point to the last character in SR$, in this case the 'A', which is not in CS$, and so may need to be scanned later. This is what line 6920 tests for.
The defined function
FN IC%(MID$(SR$, CC%, 1), CS$)
returns "true" (-1) if the character in SR$ at position CC% is in CS$. In our example here, CC% is the last character position, and its value is 'A', which is not in CS$ (currently containing whitespace). Line 6920 inverts this result (NOT FN IC%...) so that it returns "true", and sets MR% to -1. (The definition of FN IC% is given down below.)

There's also GatherWhile() and GatherUntil.

```
5399 '
       "GATHER WHILE" -- SR$(CC%..NC%) MATCHES SET CS$

5400 NF%= 0' "MATCH" CHAR SET
5410 GOSUB 5000' SCAN
5440 SC%= SL% > 0' IF SELECTION LENGTH SL% > 0, "SUCCESS"!
5490 RETURN
```

The two Gather()s are like simpler versions of the Skip()s. Like those, they set NF% (lines 5400 / 5500) and call GeneralScan() (5000). But they don't update CC% or MR%, because they're not "moving the cursor".

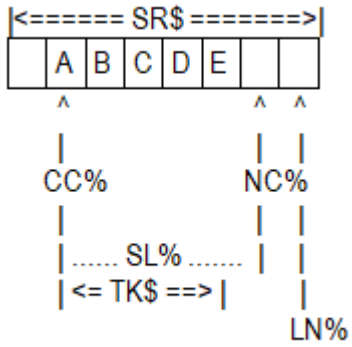
(Any code which uses a Gather() must manually update these after making use of the results: see line 9500 in GetToken() above, for example.) Instead, they report success/failure in SC% based on what the length of the "selection" SL% is. SL% gets set in GeneralScan() and is the difference between CC% and the new cursor position NC%. (If nothing was "gathered", NC% is the same as CC%, and SL% will be "false", 0.)

```
5499 '
      "GATHER UNTIL" -- SR$(CC%..NC%) UP TO CHAR IN SET CS$

5500 NF%= -1
5510 GOTO 5410' SCAN, REPORT "SUCCESS/FAILURE"
```

By this point, we have up to 2 "results":
MR% true (-1) if there's "more to scan"
SC% true (-1) if we "successfully" gathered a substring
and 4 "Scan State Variables" which tell us about the scan in progress:
LN% the length of SR$
CC% the "current cursor", where our scan currently begins within SR$
NC% the "new cursor" position after a scan
SL% the "selection length", or length of the substring that's been identified; SL%= NC% - CC%

As an example, GetToken() might be in the midst of processing the string " ABCDE ". It's already called SkipWS() and GatherUntil(). Here's what the Scan State Variables look like:



Now that we can picture them all, it`s time to revisit InitScanString() (6000), which is called whenever a new value is assigned to SR$ (for example, line 50150 in our Demo Program, above).

```
5999 '
      SET UP SCAN STATE VARIABLES AFTER NEW SOURCE$

6000 LN%= LEN( SR$): UT%= LN%: MR%= LN% > 0
6010 NC%= 0
6020 IF MR% THEN
        NC%= 1
6030 GOSUB 5900' CC%= NC%
6090 RETURN
```

(Note that these routines do not do a lot of bounds-checking: if SR$ is empty (ie, ""), line 6000 will crash. But then, if you write a program line that says

100 PRINT 5 / 0

it will crash, too: tradeoffs are always being made.)
A new Scan State Variable is introduced: "Up To", UT% defines how far along in SR$ the scan is allowed to progress, and it defaults to the end of the string. (It's possible to do scans on subsections of SR$ by resetting CC% and UT% to different values after InitScanString() is called.) Why do we even test for LN% > 0? This makes it easy to add bounds-checking on the length of SR$ if needed, and it doesn't cost much in processor time because it's only called once for every new string assignment. So long as SR$ is not empty, MR% will be "true" / -1 (there's "more to scan"), NC% will be set to the first character position, and by calling UpdateCC() (5900), we set CC% to character position 1 as well.

Each of the Skips and Gathers call a common GeneralScan() routine:

```
4999 '
     SET UP, CALL SCAN, REPORT RESULTS

5000 GOSUB 5100 ' SCAN SR$
5020 NC%= CC% + SL% ' UPDATE "NEW CURSOR"
5090 RETURN
```

A BasicScan() (5100) is performed, and we update the "new cursor", NC%.

Here's the engine of the entire string-scanning operation, BasicScan():

```
5099 '
     BASIC SCANNING ROUTINE

5100 SL%= 0
5105 FOR C%= CC% TO UT%:
     IF NOT FN XR%(  NF%,
               FN IC%( MID$( SR$, C%, 1), CS$)) THEN
                    RETURN
5110    SL%= SL% + 1: NEXT
5190 RETURN
```

First, the "selection length" SL% is initialized to 0. Next, index C% loops from the "current cursor" CC% all the way to the "up to" marker UT%, by default the end of SR$ (but BasicScan() or GeneralScan() can be called with different value for UT%).

Line 5105 should be deciphered from the "inside out". First of all, we're going to examine one character in SR$, at position C%.

MID$(SR$, C%, 1)

That single character is tested by FN IC%() to see if it's in the character set CS$ (in our demo here, the set of whitespace characters we set up in SkipWS() earlier).

FN IC%(MID$(SR$, C%, 1), CS$))

If it is in that set, FN IC%() will return "true" (-1), and our scan is "satisfied". But there's another condition to consider: the "do NOT match" Flag NF%. If NF% is set (-1), and FN IC%() reports that the single character was found in the set CS$ (-1), ie a "match", then the scan at this point is "not satisfied": "you told me not to find whitespace, but FN IC%() says this character is whitespace, so I'm not satisfied." In other words, NF% should invert the result of FN IC%().
Here's a "truth table" that illustrates what we want to see happen, with the first line depicting the example above. In line 2, NF% still requires that we "don't match" the character set, and FN IC%() replies, "Right: we didn't find it in CS$", so the scan is satisfied. Line 3: NF% now requires a match, and FN IC%() replies that it did indeed find one, so the scan is satisfied again. Finally, if NF% is not set and FN IC%() didn't find a match, the scan is also not satisfied.

"Do Not Match" flag NF%	Single Character is found in set CS$: Result of FN IC%()	Scan Satisfied?
-1	-1	
-1	0	
0	-1	
-1	0	
0	-1	
-1	0	
-1	-1	
0	0	
0	0	
0	0	

Another way to describe this is that if NF% is set (-1), the scan result is the opposite or the "inversion" of FN IC%(); but if NF% is not set (0), the result is just the whatever FN IC%() reports.
We could have coded this as:

```
5107    IF NF% THEN
        IF FN IC%(...) THEN
            RETURN
5108    IF NOT FN IC%() THEN
        RETURN
```

Instead, we're using XOR ("exclusive OR"), which only results in true (-1) if its two arguments don't match.

Argument 1	Argument 2	XOR result
-1	-1	0
-1	0	-1
0	-1	-1
0	0	0

This is the effect of the defined function FN XR%():

```
FN XR%(     NF%,
        FN IC%( MID$( SR$, C%, 1), CS$))
    )
```

The result of FN XR%() here tells us if the scan is "satisfied", which is not the same as "matching". For example, if we want to SkipWS() (see above), we leave "argument 1", NF%, reset (0). If FN IC%() finds that the character at position C% matches a member of the set CS$, then "argument 2" is true (-1), and the XOR result is "satisfied" (-1).
However, if we want to GatherUntil() (see above), we set argument 1, NF%, as -1, "don't match", and if the character at position C% does not match a member of the set CS$ (whitespace in our demo program), then argument 2 is 0, and the XOR result is "true" (-1), again "satisfied".
If FN XR%() is satisfied, then the "selection length" SL% gets incremented (line 5110), and we go on to test the next character (NEXT). If it's not, we immediately leave this subroutine: it's done its work. Therefore, the full test to RETURN in the second part of line 5105 is:

```
IF NOT FN XR%(  NF%,
            FN IC%( MID$( SR$, C%, 1), CS$)) THEN
                RETURN
```

All that BasicScan() does, then, is to process SR$ one character at a time, from position CC% to UT%, incrementing SL% each time the XOR test is satisfied, or exiting as soon as it's not.
Here's the function definitions for FN IC%() and FN XR%():

```
40010 DEF FN XR%( A%, B%)= (A% OR B%) - (A% AND B%)' XOR
( A%, B%)
40030 DEF FN IC%( CH$, ST$)= INSTR( ST$, CH$) > 0' T IF CH$ IN SET
ST$
```

The demo program "Main()" in line 50000 calls the subroutines and functions described here like this:

```
Main() / 50000
      DefineFunctions() / 40000
      InitScanString() / 6000
      GetToken() / 9000
            SkipWS() / 7000
                  SkipWhile() / 5200
                        GeneralScan() / 5000
                              BasicScan() / 5100
                                    FN IC%()
                                    FN XR%()
                              UpdateCursor() / 5900
                              IsThereMoreToScan() / 6900
            GatherUntil() / 5500
                  GeneralScan() / 5000
                              BasicScan() / 5100
                                    FN IC%()
                                    FN XR%()
                              UpdateCursor() / 5900
                              IsThereMoreToScan() / 6900
            UpdateCursor() / 5900
```

and here's the entire demo program listing:

```
10 GOTO 50000'

      SCANARTC/BAS -- STRING SCANNING DEMO
                  JAN 31, 2023

4999 '
      GENERAL SCAN

5000 GOSUB 5100' SCAN SR$
5020 NC%= CC% + SL% ' UPDATE "NEW CURSOR"
```

```
5090 RETURN
5099 '
     BASIC SCANNING ROUTINE

5100 SL%= 0
5105 FOR C%= CC% TO UT%:
     IF NOT FN XR%( NF%, FN IC%( MID$( SR$, C%, 1), CS$)) THEN
          RETURN
5110   SL%= SL% + 1: NEXT
5190 RETURN
5199 '
     "SKIP WHILE" ("IGNORE") --
          SKIP UNTIL NOT IN CS$, UPDATE CC%, MR%

5200 NF%= 0' "MATCH" CHAR SET
5210 GOSUB 5000' SCAN
5220 GOSUB 5900' UPDATE CC%
5230 GOSUB 6900' DETERMINE IF THERE'S "MORE" TO SCAN
5290 RETURN
5299 '
     "SKIP UNTIL" ("FIND") --
          FIND 1ST MATCH W/ CS$, UPDATE CC%, MR%

5300 NF%= -1' "DON'T MATCH" CHAR SET
5310 GOTO 5210' SCAN; UPDATE CC%; DETERMINE IF "MORE" TO SCAN
5399 '
     "GATHER WHILE" -- SR$(CC%..NC%) MATCHES SET CS$

5400 NF%= 0' "MATCH" CHAR SET
5410 GOSUB 5000' SCAN
5440 SC%= SL% > 0' IF SELECTION LENGTH SL% > 0, "SUCCESS"!
5490 RETURN
5499 '
     "GATHER UNTIL" -- SR$(CC%..NC%) UP TO CHAR IN SET CS$

5500 NF%= -1
5510 GOTO 5410' SCAN, REPORT "SUCCESS/FAILURE"
5899 '
     UPDATE CC%= NC%

5900 CC%= NC%: RETURN
5999 '
     SET UP SCAN STATE VARIABLES AFTER NEW SOURCE$

6000 LN%= LEN( SR$): UT%= LN%: MR%= LN% > 0
6010 NC%= 0
6020 IF MR% THEN
```

```
      NC%= 1
6030 GOSUB 5900' CC%= NC%
6090 RETURN
6899 '
      DETERMINE IF THERE'S "MORE" TO SCAN

6900 MR%= CC% < LN%' IF WE'RE NOT AT END OF SR$, THEN "MORE"
6920 IF NOT MR% THEN
      MR%= NOT FN IC%( MID$( SR$, CC%, 1), CS$)
      ' IF CHAR AT CC% NOT IN CS$, THEN "MORE"
6990 RETURN
6998 '
      SOME USEFUL SCANNING ROUTINES:

6999 '
      1. SKIP WS

7000 CS$= " " + CHR$( 10) + CHR$( 13) ' WHITESPACE CHARACTERS
7050 GOSUB 5200' "SKIP WHILE" OR "IGNORE" WS
7090 RETURN
7999 '
      2. NEXT TOKEN -- SKIP WS, RETURN TOKEN AFTER TOKEN
                        UNTIL MR% IS F

9000 '
9010 TK$= ""' START WITH EMPTY TOKEN$
9020 SC%= 0' "RESULT" (IE, SUCCESS/FAILURE)
9100 GOSUB 7000' SKIP WS
9110 IF NOT MR% THEN GOTO 9500
9120   GOSUB 5500' "GATHER UNTIL" CS$ (WS)
9130   IF SC% THEN
            TK$= MID$( SR$, CC%, SL%)
9500 GOSUB 5900' UPDATE CC%= NC%
9990 RETURN
40000 '
      INITIALIZATION

40010 DEF FN XR%( A%, B%)= (A% OR B%) - (A% AND B%)' XOR
( A%, B%)
40030 DEF FN IC%( CH$, ST$)= INSTR( ST$, CH$) > 0' T IF CH$ IN SET
ST$
40090 RETURN
49999 '
      *** STRING SCANNING DEMO
      ***     MAIN PROGRAM

50000 CLEAR 500
```

```
50010 CLS: PRINT "STRING SCANNING DEMO:"
50020 PRINT "PARSE 'COMMAND LINE' INTO AV$(),"
50030 PRINT "TOTALLING ARGUMENTS IN AC%"
50049 '
      INITIALIZE:

50050 GOSUB 40000
50059 '
      INITIALIZE "ARGC" / AC% TO INDICATE
            "NO TOKENS YET" / -1

50060 AC%= -1
50070 SR$= ""
50099 '
      PROMPT FOR TEST STRING:
            REJECT ANY EMPTY ""

50100 PRINT: PRINT "ENTER STRING, IE 'COMMAND LINE (PROGRAM
ARGS... )'"
50110     LINE INPUT SR$
50120     IF SR$ = "" THEN GOTO 50100
50149 '
      SET UP SCANNING VARIABLES FROM NEW SR$

50150 GOSUB 6000
50199 '
      SCAN & RETRIEVE TOKENS LOOP:

50200 GOSUB 9000
50210     IF SC% THEN
            AC%= AC% + 1:
            AV$(AC%)= TK$
            ' SUCCESS: SAVE TOKEN IN AV$()
50220     IF MR% THEN
            GOTO 50200
            ' MORE TOKENS TO SCAN
50299 '
      IF ANY TOKENS GATHERED, PRINT THEM OUT:

50300 IF AC% < 0 THEN
            PRINT "NO TOKENS SCANNED":
            GOTO 51000
50309 '
        HEADER

50310     PRINT
50320     PRINT "AC%:"; AC%
```

```
50330     PRINT "TOKEN #", "TOKEN"
50340     PRINT "-------", "-----"
50399 '
          PRINT OUT ALL TOKENS

50400     FOR T%= 0 TO AC%
50410          PRINT T%, AV$(T%)
50420          NEXT
50999 '
          DONE

51000 PRINT
51010 GOTO 50060
```

Some notes on the listing (for the curious):

Why immediately GOTO 50000? This is my usual BASIC program layout. It's kind of like a Pascal program (remember those?)--declarations, functions and procedures first, then the Main() program. In the case of TRS-80 BASIC, I like to list the most-used subroutines first, because the interpreter will search for and find those lines the quickest.

Having said that, is this the quickest implementation? Hardly. Here's a quicker version of GeneralScan() and BasicScan() (lines 5000..5190) combined:

```
5000 SL%=0:FORC%=CC%TOUT%:IFFNXR%(NF%,INSTR
(CS$,MID$(SR$,C%,1))>0)THENSL%=SL%+1:NEXT
5002 NC%=CC%+SL%:MR%=NC%<=LEN(SR$):RETURN
```

Why did I develop these routines? Firstly, for the fun of it. Secondly, to prototype routines I hope eventually to write in Z-80 Assembler.

Jens Guenther has updated his fork of the SDLTRS Emulator to version 1.2.26. Changelog includes:

- Added Sector Size Support for WD1000/1010
- Added Hard Disks (WD) for EACA EG 3200 Genie III and TCS Genie IIIs
- Added Floppy Disk controller (Expansion Interface) option
- Added debug command to modify the ROM
- Fixed Aster CT-80 charset in TRS-80 mode
- Fixed TRS-80 Model 4/4P Video Page

You can download the most recent SDLTRS II from the Gitlab page: https://gitlab.com/jengun/sdltrs.

2022 POWER SUPPLIES
FOR THE TRS-80 MODEL 3 / MODEL 4/4D/4P

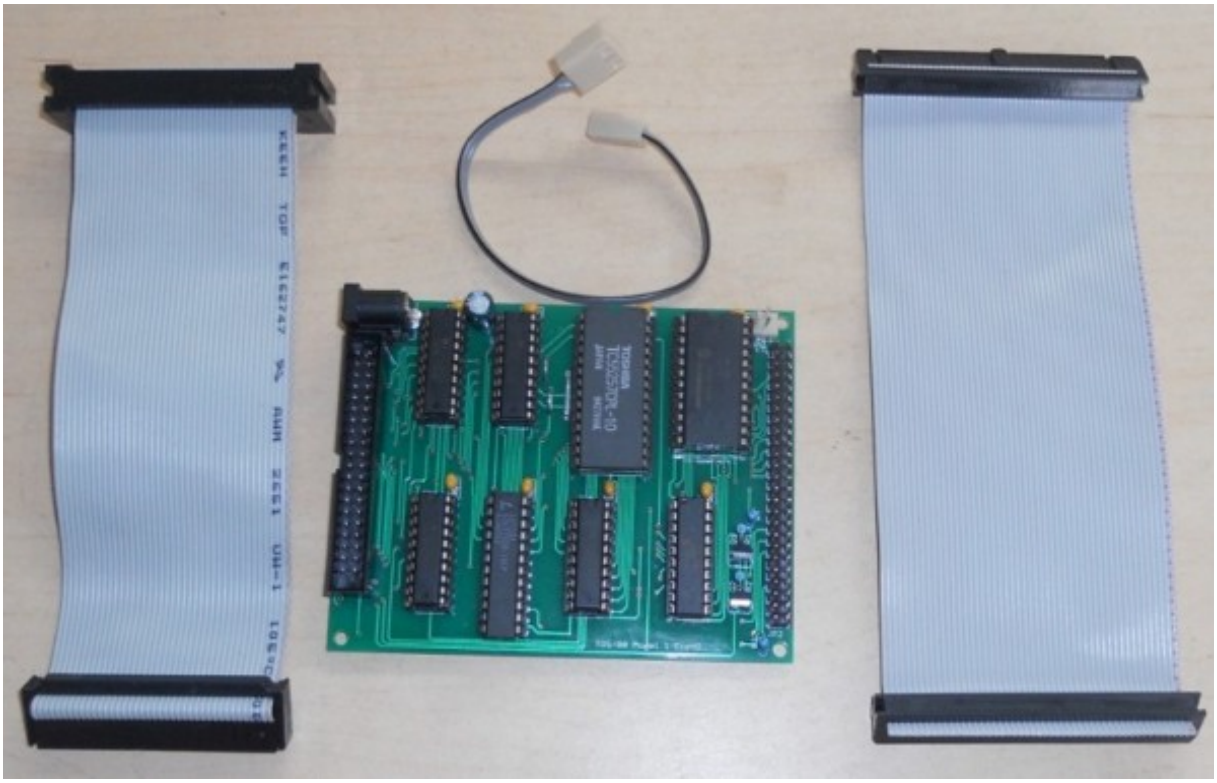
* Uses modern day MEAN WELL power supply units
* 50w for Model 3 version and 65w for the Model 4 version
* The Model 4 version uses a SEPARATE +12V POWER SUPPLY (prevents video "spasm")
* No more RIFA CAP explosions
* No more COLD solder joints
* Units are DIRECT REPLACEMENTS for the original power supplies in all models
* Extra +5V output connectors (1 on Model 3 and 2 on Model 4) for Gotek/freHD boards
* For more information and pricing contact

newSOFT

https://jaynewirth.wixsite.com/newsoft (website)
jaynewirth@verizon.net (email)

“Quinnterface” Mini Expansion Interface for 16K Model 1 FreHD users.

- COMPLETLY ASSEMBLED AND TESTED -



This is THE perfect device for all Model 1 users who own a 16K Level II unit, (which is most of us!), but no Expansion Interface or disk drives, especially if you don't want to modify your M1 with upgraded boot ROM or memory upgrade.

The ‘Quinnterface’, developed by J. Andrew Quinn from New Zealand, adds 32K RAM and auto-boot functionality to you FreHD.

U.K. And Europe, contact Bas. at BetaGamma Computing.
U.S.A. And rest of the world, contact Mav. At “The Right Stuff”

LOOKING FOR FAST, INEXPENSIVE, UNLIMITED MASS STORAGE FOR YOUR TRS-80 MODEL I/III/4/4P/4D?

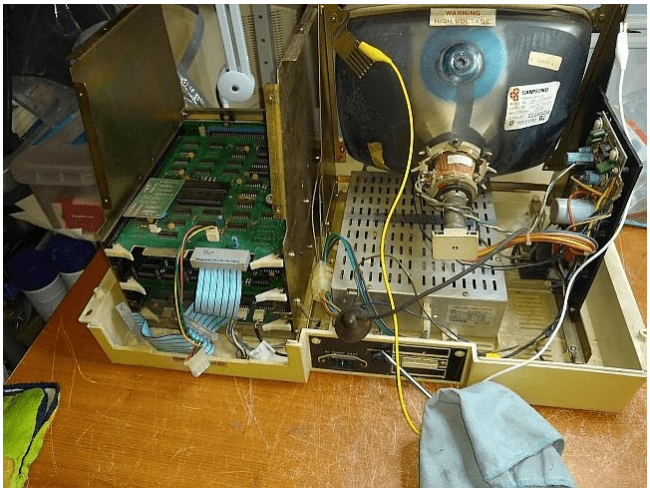
The amazing

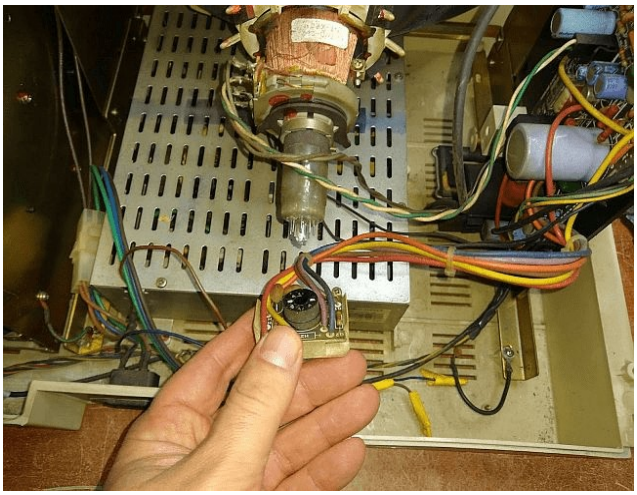
"FreHD"

-Emulates a TRS-80 hard drive, but faster than any hard drive!
-Works with your favourite DOS (LS-DOS, LDOS, CP/M, Newdos/80 2.5)
-Uses SD card for storage medium
-Bonus free Real Time Clock function!
-Designed in Belgium and proudly built and shipped from Australia
-Kit form or fully assembled

Order yours today
http://members.iinet.net.au/~ianmav/trs80/

HERE'S A SIGHT FOR SORE EYES!
THIS IS **KRIS GARREIN'S** VIDEO GENIE III
AFTER A LOT OF WORK





NEWS FROM FRANCE

Liberté • Égalité • Fraternité

PASCAL HOLDRY

Bonjour Dusty, Bonjour à tous,

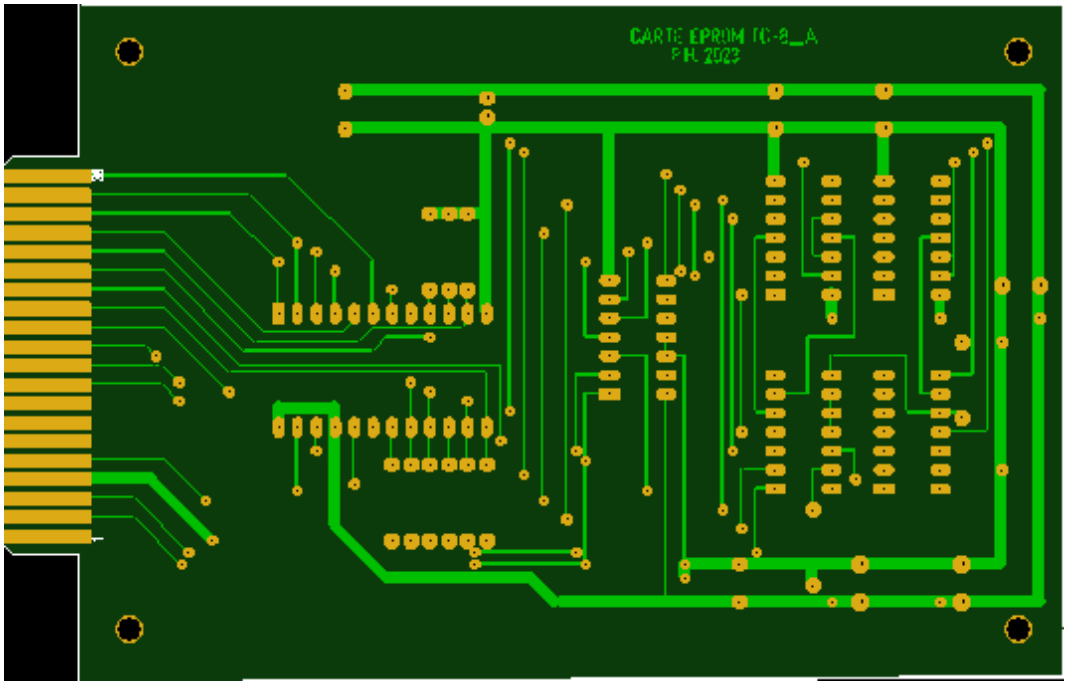
I just made the clone card of the TC-8 card which allows to read the cassettes with X5 speed. The TC-8 has almost the same functions as the "EXATRON", but with traditional audio cassettes.

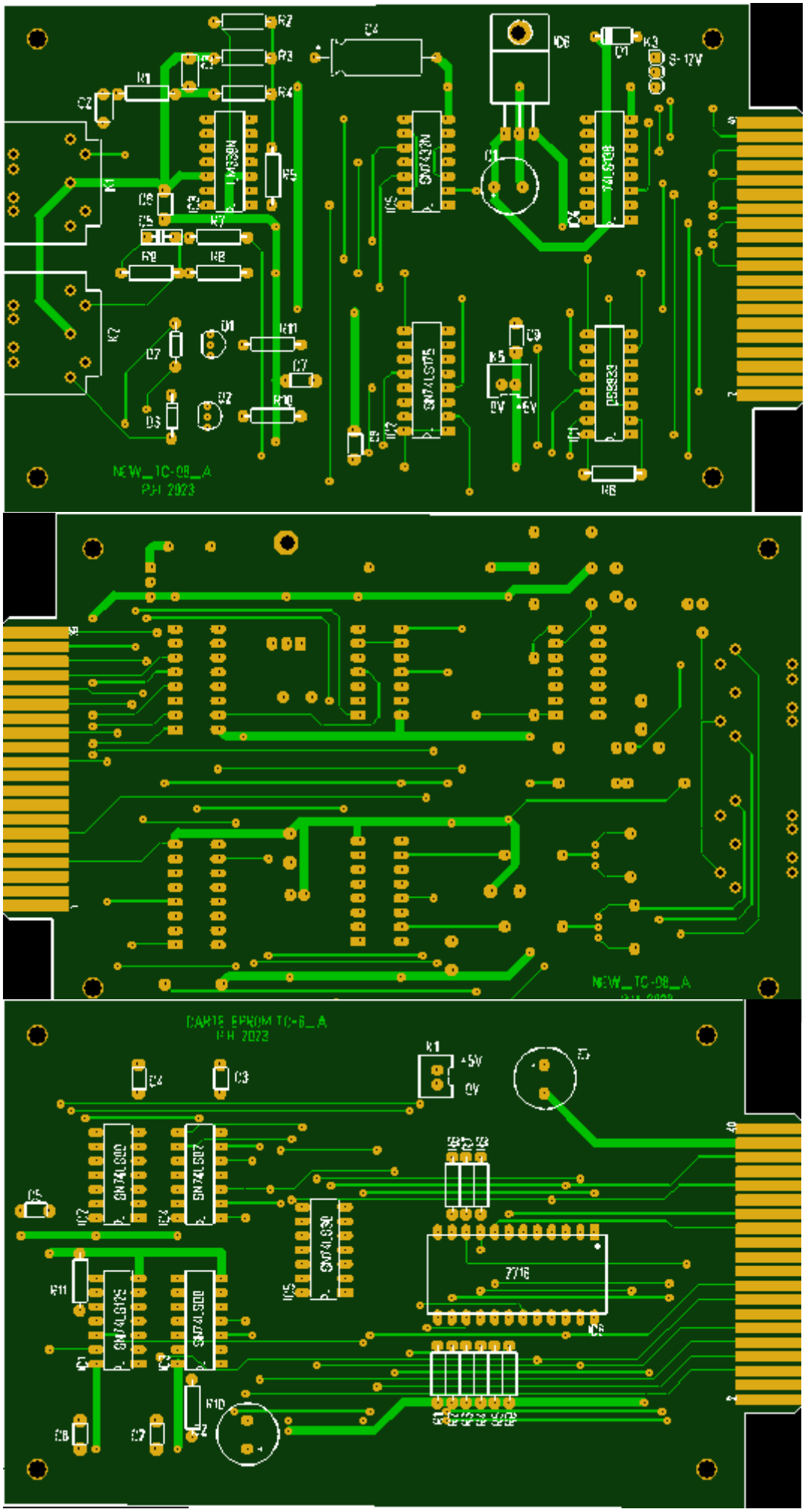
This TC-8 was lent to me by Stephen. The TC-8 seems to have been made in Great Britain. Hans had sent me a TC-8 clone a few years ago This TC-8 has been tinkered with an additional card with an Eprom.

Unfortunately I don't have the cassette sold with the TC-8 box, I'm looking for this cassette.
Maybe the Eprom allows to use part of the TC-8, but I would like to have a copy of the original program

Dusty can you help to find a copy of the cassette?

Sure can Pascal, if anyone has a copy of the necessary TC-8 software, please contact me and I'll pass on your details to Pascal.




NEW_TC-08_A
PH 2023
NEW_TC-08_A
CARTE EPROM TC-8_A
PH 2023
+5V
0V
7716

"VOXBOX"

Voice Recognition Unit for TRS-80 Model 1

Pascal Holdry

The TRS-80 VOXBOX voice recognition unit allows you to take advantage of this new technology on your Level II TRS-80 computer. You can now use words and phrases to control and instruct your computer and to enter data. Focus on the work or game at your fingertips, like watching video, without the distraction of typing on the keyboard.

- Simple to use, connects directly to TRS-80 or TRS-80 Bus Card Edge expansion interface.
- Includes a dynamic push-to-talk microphone with coiled cord for communication with the VOXBOX.
- Includes a machine language "driver" program and three application programs.
- Using the examples and instructions in this manual, you can write your own programs for custom applications.
- Use the learning mode to teach the VOXBOX up to 32 words or phrases, each word/phrase lasting up to 1.2 seconds.
-

Re-train it at any time to recognize up to 32 different words in any language!

Note: Speech recognition is a new technology. In fact, your Radio Shack VOXBOX is one of the first devices of its kind to be both available and affordable to the general public.

For this reason, Radio Shack recommends that the device be used primarily for entertainment and experimentation. Proceed with caution before engaging the unit in serious application.

Equipment required

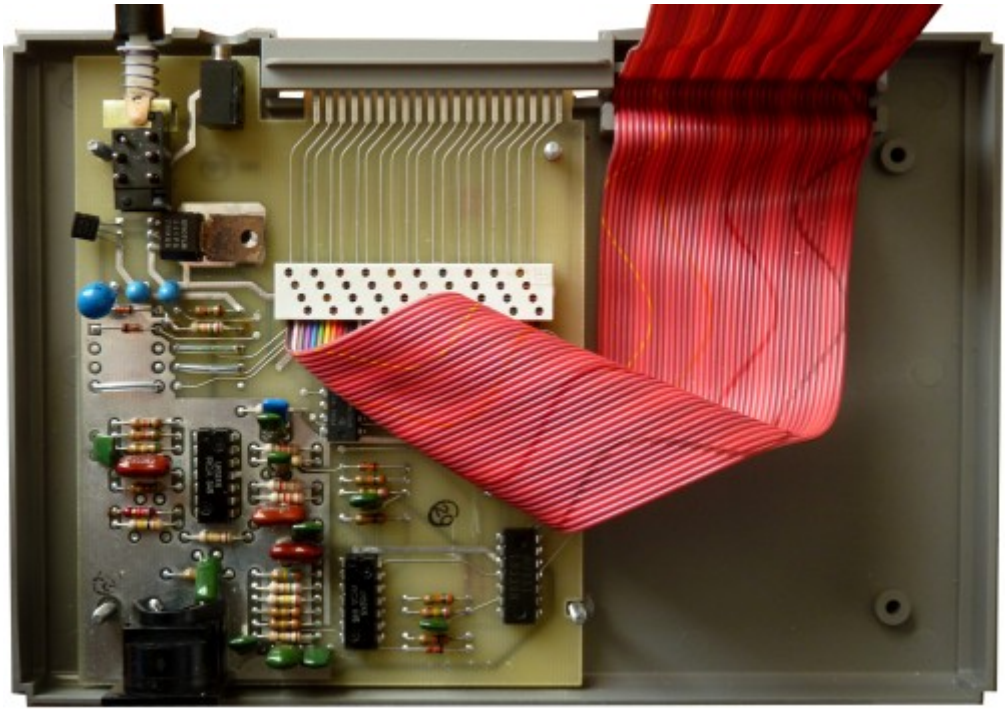
TRS-80 with 16K RAM
Tape recorder
Optional expansion interface

This box was given to me by Ian Mavric, whom I sincerely thank.

View with the microphone

As you can see the case does not completely close, Tandy had forgotten the thickness of the "PCB" when creating the molds to build its case

View of the inside of the box



You can see that the bottom right screw is not included in its entirety in the "PCB", a notch in the "PCB" replaces a drill hole.

View of the back of the box

On this photo you can better see the design error of where Tandy had forgotten the thickness of the "PCB" when creating the molds to build this case.

RE-VOX_BOX

by Pascal Holdry

Aim:

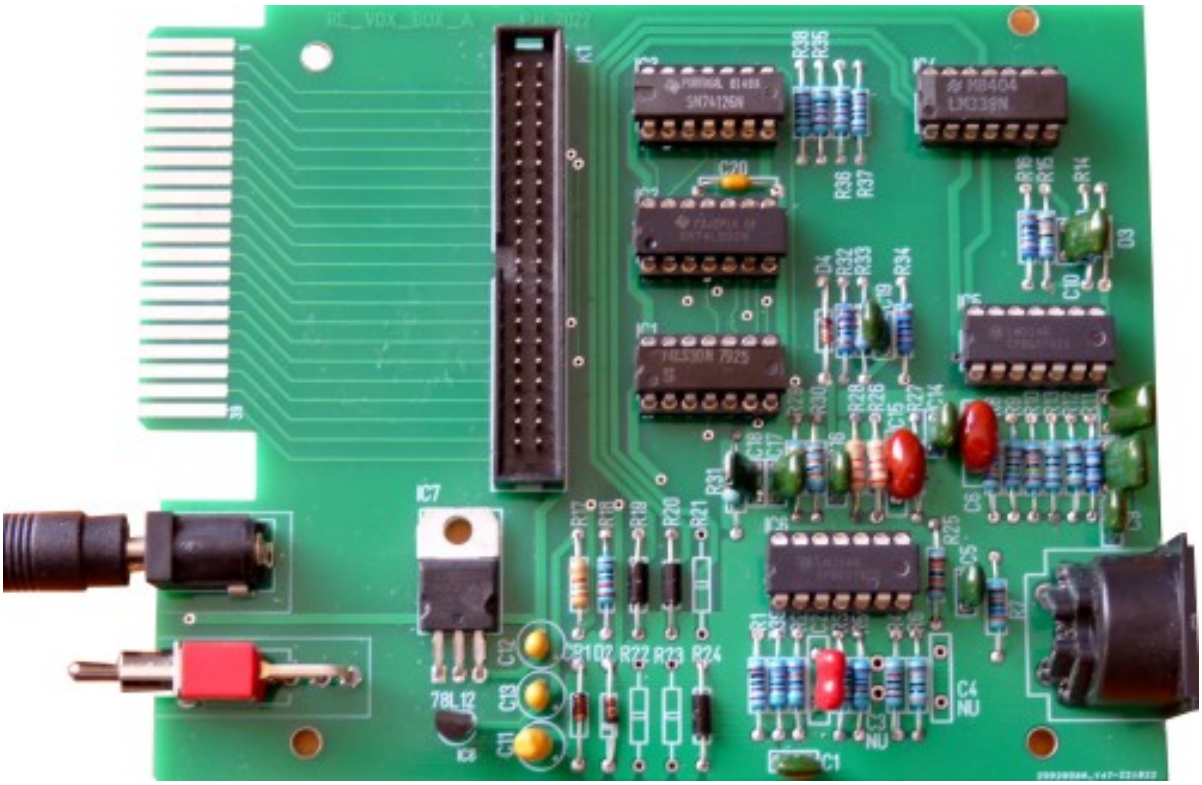
1. Being able to recover existing software
2. Keep track of our digital heritage
3. Provide a schematic for troubleshooting and understanding how the board works.

Specifications:

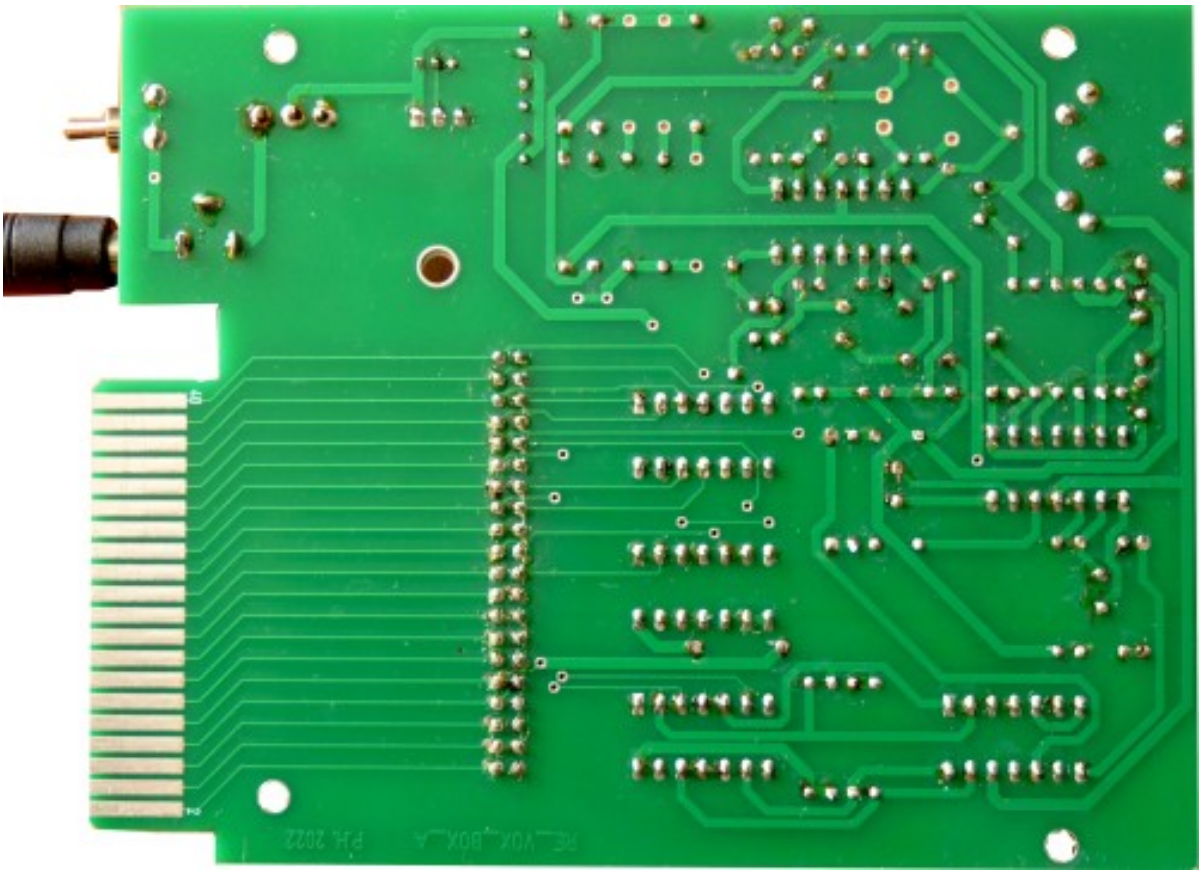
1. Be as compatible as possible with the VoxBox box
2.

Warning: This card is version A and contains an error for the installation of the DIN socket (holes too small and footprint only allowing the installation of a single type of DIN socket), a version B is in production correcting these errors and allowing the installation of two types of DIN socket

Top view



Bottom View



The RE-VOX_BOX voice recognition unit allows you to take advantage of old technology on your TRS-80 M1. You can now use words and phrases to control and instruct your computer and to enter data. Focus on handy work or play, like viewing video, without the distraction of typing on the keyboard. Simple to use, connects directly to the TRS-80 or USB interface TRS-80 Bus Card Edge expansion.

- 1) Simple to use, connects directly to TRS-80 or TRS-80 Bus Card Edge expansion interface.
- 2) Using the examples and instructions in this manual, you can write your own programs for custom applications.
- 3) Use learning mode to teach the RE-VOX_BOX up to 32 words or phrases, each word/phrase can last up to 1.2 seconds.
- 4) Re-train it anytime to recognize 32 different word vocabulary in any language!

Note: This voice recognition is old technology. In fact, your RE-VOX_BOX is one of the first devices of its kind for the general public. For this reason, I recommend that the device be used primarily for entertainment and experimentation. Proceed with caution before engaging the unit in serious application.

Equipment required

1. TRS-80 with 16K RAM
2. Sound Recorder
3. Optional expansion interface

Please note the RE-VOX_BOX and the VOXBOX were designed to operate mainly on magnetic tapes, but there is a program that I discovered on the WEB "VXDRIVER/CMD" which allows these cards to operate under LDOS 5.3.1..

This program replaces the original programs SP16, SP32, SP48.

To make this card work under floppy and 48K of memory

1. You run: LDOS 5.3.1
2. Then: VXDRIVER/CMD
3. Then: BASIC (MEM=61439)
4. Then your program

YOU choose which emulation system you want.
Out of:

BBC B
TRS-80 Model I/III/IV
ZX81

State your choice (and choice of keyboard) in "message to seller" upon purchasing.

Whilst we keep all parts in stock we build to order, so allow up to five days for dispatch.
(It's usually done in three.)

ALL systems are preloaded with hundreds if not thousands of games from back in THE day.
And load instantly. (No cassettes!)

Some games support any standard USB controller which you can plug straight into your RX-1000. Keyboard controls are identical to the original now vintage computers of our childhoods.

FREE postage to UK buyers.

Place your order and get ready for the adventure.
Your childhood/youth in a box.
Plug and play.

All software included is in the public domain aside from RX-1000 operating systems which are (C) 2022 Retro Shack.

YouTube link below is to a short promo video made when the RX-1000 was developed.

https://youtu.be/eRB8T4bNv6Q

I THOUGHT UP A SILLY LITTLE TRS-80 JOKE WHICH I DECIDED TO PUT IN THE FORM OF A BASIC PROGRAM. THE PUNCH LINE IS SCRAMBLED JUST ENOUGH TO KEEP FROM GIVING IT AWAY.

GIVEN THE QUALITY OF THE JOKE I DEEMED IT NECESSARY FOR THE PROGRAM TO DO ITS OWN "LAUGH TRACK" IN SUPPORT.

-- GEORGE --


```
READY
>@LOAD"GEOSJOKE/BAS"
>LIST
10 CLS:PRINT"WHY ARE TRS-80 GRAPHICS SO COOL?"
20 IFINKEY$=""THEN20
30 A$="CFIDBVTF!JUT!QJYFMT!BSF!OPU!TRVBSF!"
40 PRINT:PRINT" ";
60 FORX=1TOLEN(A$):PRINTCHR$(ASC(MID$(A$,X,1))-1);
70 SET(2*X,4):SET(2*X,9):NEXT
80 FORY=4TO6STEP2:SET(0,Y):SET(2*X,Y):SET(0,Y+3):SET(2*X,Y+3):NE
XT
90 P=256+RND(764):PRINT@P,"HA!";:
100 FORX=0TO50:NEXT:PRINT@P,"   ";:GOTO 90
READY
>_
```

HTTPS://WWW.YOUTUBE.COM/WATCH?V=ZE3NMXTLGJK

Thirsting for adventure?
Brass lantern getting dusty?

Take your Model 1, 3 or 4 on exciting new adventures never available until now....

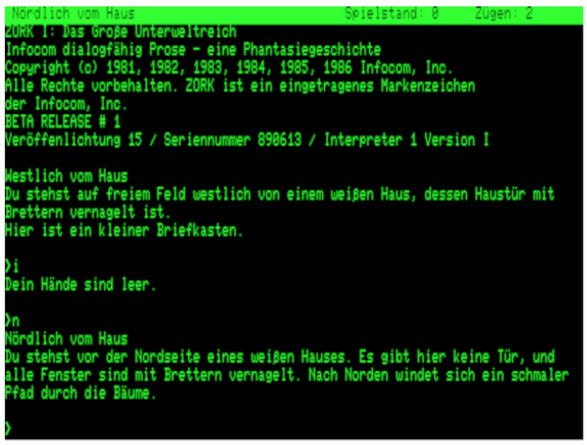
M4ZVM enables your TRS-80 Model 1,3 or 4 to play Infocom and post-Infocom adventures, taking full avdantage of your hardware. Supports old favourites like Zork and Seastalker (including Sonarscope), as well as new-to-Model-4 games including Trinity, Border Zone and Sherlock, plus hundreds more available in the Interactive Fiction archive.

M4ZVM is available for download from https://gitlab.com/sijnstral/m4zvm (includes feature details and source)
Enhanced versions ready for download!

*128K recommended

Actual Screenshots of games new to the model 4, running M4ZVM (on trs80gp)

Beyond Zork



Zork 1 (German)

M4ZVM features include:
* z1-z8 game execution (z6 support is experimental)
* named game save/load
* accented characters
* 128K recommended - NEW Minimised 64K version
* Optionaly set the infamous "Tandy bit"
* NEW Model 1 and Model 3 versions

One of the most infuriating features of the Tandy Level II manual is that there is no index. We now remedy that with contents you can cut out and stick into your manual.

TRS-80 Level II Index

Reprinted from Orange County TRS-80 Users' Group Newsletter, available by donation, £10 a year from user group, 24232 Tahoe ct., Laguna Niguel CA 92677.

PRACTICAL COMPUTING *April 1979*

Serial-printer drive routine for Tandy TRS-80

ONE OF the attractive features of the TRS-80, after the pleasure of finding that you have 16K of usable RAM and 12K of Basic for less than the price of an 8K Brand X, is the ease with which a printer can be driven.

There is not much doubt that a printer is the first essential accessory, assuming that you start with a reasonable amount of RAM. The low-price printers, such as the Nascom Imp or the Epson are serial printers. The natural companion to the TRS-80 is the Centronics, which uses a parallel port, but the Centronics costs considerably more than a complete TRS-80, and if it happens to carry the Radio Shack label, it may cost even more.

Fortunately, serial printers can be interfaced to the TRS-80 through the cassette port, using a very simple interface consisting of little more than a 741 integrated circuit, a rectifier bridge, and a few resistors and capacitors. The total cost of the interface, if you build it yourself, is about £2.

If you are not in the hardware business, however, you will be asked for about £40

by Ian Sinclair

for an interface which presumably must use a very expensive plug and socket. Some of the cost of the commercial effort is the software which accompanies it, because the use of the cassette port for serial printer output is not one of the considerations which Tandy had in mind when it assembled its package.

There are many serial-printer drive programs, ranging from a bare 100 bytes all the way to fully-fledged printer, screen and keyboard controllers, like the excellent KVP Extender from Microcomputer Applications.

I wanted more than a straightforward routine, because I like to keep listings as I develop programs and nothing is more awkward than keeping long rolls of paper. Since I keep the original program notes on A4 and in A4 files, it seems logical to keep the listings there as well, which means that I needed a left-hand margin, to allow room for the punch-holes.

As the Nascom Imp printer starts well over to the left-hand side on listings, that obviously was a software task. In addition, using A4 files meant that the listing should be in groups of 66 lines, so that I could divide the Teletype roll into pages. That in turn meant a lines-per-page counter in software which could be re-set so that I could set top-of-page when I started a listing, instead of having to accept a random number of lines before a page break.

At that point in the specification, a few bells and whistles started to be added. I wanted to be able to slow down the listing on the screen with a simple command — I hate having to poke numbers into memory — just as that other computer does, and I wanted the usual keyboard delay because my TRS-80 is an old one with keybounce.

To make the specification more awkward, I originally wanted the program re-locatable, so that I could place it anywhere in memory. There was only one way of obtaining all this, and that was to write the program myself.

Since the TRS-80 uses the Z-80 CPU, there is a version of the Zen assembler-editor-debugger available for it. If there are still people who run Z-80 machines and have not encountered Zen, I believe it is the best editor/assembler package available.

It is simple to use — which means that you can stop thumbing through the instructions after a few hours, which means I make for more use of it than of the package I used before.

As an assembler/editor, it is very versatile, allowing quick and easy assembler program writing without having to watch the size of spaces or the length of label-names; editing is very simple. It even permits assembler programs to be combined from tape, and, of course, writes object code tapes in TRS-80 standard code.

In addition, Zen has full monitor facilities, allowing you to read machine-code

Figure 1.

```
 1                      ;LPRINT ROUTINE WITH LHS MARGIN
 2                      ;66 LINES/PAGE AND 72 CH/LINE
 3                              ORG   7F11H
 4                              LOAD  6F00H
 5                      CHRSLN: EQU   4029H
 6                      LNFCE:  EQU   4028H
 7 7F11 E5                      PUSH  HL
 8 7F12 21347F                  LD    HL,LPRT
 9 7F15 22C640                  LD    (40C6H),HL
10 7F18 21FC7F                  LD    HL,KBFIX
11 7F1B 221E40                  LD    (401EH),HL
12 7F1E 2ADE7F                  LD    HL,(CHRLN)
13 7F21 222940                  LD    (4029H),HL
14 7F24 21DF7F                  LD    HL,TOPPAG
15 7F27 227D41                  LD    (417DH),HL
16 7F2A 21E77F                  LD    HL, DELAY
17 7F2D 228341                  LD    (4183H),HL
18 7F30 E1                      POP   HL
19 7F31 C37200                  JP    0072H
20 7F34 F3                      DI
21 7F35 79                      LD    A,C
22 7F36 F5                      PUSH  AF
23 7F37 D5                      PUSH  DE
24 7F38 FDE5                    PUSH  IY
25 7F3A CD3300                  CALL  0033H
26 7F3D FDE1                    POP   IY
27 7F3F D1                      POP   DE
28 7F40 F1                      POP   AF
29 7F41 FE0D                    CP    0DH
30 7F43 2805                    JR    Z,RELOAD
31 7F45 FE20                    CP    20H
32 7F47 D8                      RET   C
33 7F48 181C                    JR    START
34 7F4A E5                      PUSH  HL
35 7F4B F5                      PUSH  AF
36 7F4C 21DE7F                  LD    HL,CHRLN
```

(continued on next page)

tapes, examine the contents, change code, copy tapes, copy code from one memory location to another — which is why I like re-locatable programs — and it can even load code into memory as it is assembled, without having to make a tape first.

I used Zen to create the assembler program of figure 1 which is why the line numbers follow the 1,2,3 order and are placed at the start of each line, rather than the convention used by Radio Shack. Those line numbers, incidentally, are not recorded when you tape an assembler program; they are allocated by Zen when the program is loaded, which is why programs can be merged, no matter what line numbers they used.

It does mean, though, that the Zen assembler tapes are not compatible with tapes created by a disassembler operating to Radio Shack standards.

The program is not annotated. Notes occupy a good deal of memory and tape, and I prefer to keep copious notes on a separate sheet. For the benefit of anyone who wants to modify the program either for the TRS-80 or for another Z-80 based machine, here is an outline of the program operation.

The baud rate is set at 300, since that is the maximum rate recommended for a serial printer operated over a single line, with no provision for handshake — no signals can pass back from the printer to the computer to control the rate of transmission.

Program origin

The origin of the program as shown is 7F11H, 32529D, and two of the printer-control RAM addresses in the TRS-80 are used. They are, of course, not part of the 16K user RAM, but a separate section of about 3K of RAM which can be used for a remarkable range of control operations, and which are an Aladdin's cave for machine-language programmers.

Location 4029H (16525D) is used to store the number of characters per line, and 4028H (16524D) is used for the number of lines per page. Four addresses in the program space itself are also used, and they can be Poked so as to alter the print format as desired. As set up, the program prints 72 characters per line and 66 lines per page, with six blanks between pages to allow for cutting.

The first section of the program from 7F11H to 7F2FH loads the locations in low RAM. 4026H is the address for the start of the printing routine. The address which is normally loaded into 4026H, 4027H is 058DH, the ROM routine for a parallel printer, and we have to substitute 7F34H, so that the LPRINT and LLIST commands of the TRS-80 direct the print commands to our routine starting at 7F34H. 401EH is the start address location for the screen-write routine, normally 0458H, but now changed to 7FF2H.

That allows us to put a delay into the screen-print routine, and performs two useful functions — it allows slow-listing to be controlled, and it also debounces the wretched keyboard of my old TRS-80 — new models seem to have a new keyboard with no bounce problems. The number of characters per line is loaded into 4029H; no figure is loaded into 4028H because it is set at 66 lines per page by the TRS-80

(continued from previous page)

```
 37 7F4F 3A2940          LD    A,(CHRSLN)
 38 7F52 77              LD    (HL),A
 39 7F53 21DC7F          LD    HL,LNCNT
 40 7F56 35              DEC   (HL)
 41 7F57 200B            JR    NZ,RUN
 42 7F59 2A2840          LD    HL,(LNPGE)
 43 7F5C 22DC7F          LD    (LNCNT),HL
 44 7F5F 21DD7F          LD    HL,ENDPAG
 45 7F62 3606            LD    (HL),06H
 46 7F64 F1      RUN:    POP   AF
 47 7F65 E1              POP   HL
 48 7F66 F5      START:  PUSH  AF
 49 7F67 E5              PUSH  HL
 50 7F68 C5              PUSH  BC
 51 7F69 0609    AGAIN:  LD    B,09H
 52 7F6B 37              SCF
 53 7F6C F5              PUSH  AF
 54 7F6D F5              PUSH  AF
 55 7F6E 2101FC          LD    HL,0FC01H
 56 7F71 CD2102          CALL  0221H
 57 7F74 CD797F          CALL  DLY
 58 7F77 1809            JR    ROUND
 59 7F79 21DE00  DLY:    LD    HL,00DEH
 60 7F7C 2B      DLY1:   DEC   HL
 61 7F7D 7C              LD    A,H
 62 7F7E B5              OR    L
 63 7F7F 20FB            JR    NZ,DLY1
 64 7F81 C9              RET
 65 7F82 F1      ROUND:  POP   AF
 66 7F83 1F              RRA
 67 7F84 F5              PUSH  AF
 68 7F85 3005            JR    NC,OUT1
 69 7F87 2102FC          LD    HL,0FC02H
 70 7F8A 1805            JR    OUT2
 71 7F8C C600    OUT1:   ADD   A,00H
 72 7F8E 2101FC          LD    HL,0FC01H
 73 7F91 CD2102  OUT2:   CALL  0221H
 74 7F94 CD797F          CALL  DLY
 75 7F97 10E9            DJNZ  ROUND
 76 7F99 2102FC          LD    HL,0FC02H
 77 7F9C CD2102          CALL  0221H
 78 7F9F CD797F          CALL  DLY
 79 7FA2 F1      OUT3:   POP   AF
 80 7FA3 F1              POP   AF
 81 7FA4 FE0D            CP    0DH
 82 7FA6 2022            JR    NZ,TEST2
 83 7FA8 21DD7F  TEST1:  LD    HL,ENDPAG
 84 7FAB AF              XOR   A
 85 7FAC B6              OR    (HL)
 86 7FAD 2812            JR    Z,BLNK
 87 7FAF 35              DEC   (HL)
 88 7FB0 3E0D            LD    A,0DH
 89 7FB2 1805            JR    AGAIN
 90 7FB4 C1      OUT4:   POP   BC
 91 7FB5 21DE7F          LD    HL,CHRLN
 92 7FB8 35              DEC   (HL)
 93 7FB9 201D            JR    NZ,OUT5
 94 7FBB E1              POP   HL
 95 7FBC F1              POP   AF
 96 7FBD 3E0D            LD    A,0DH
 97 7FBF 1809            JR    RELOAD
 98 7FC1 21DB7F  BLNK:   LD    HL,LHSPCE
 99 7FC4 3603            LD    (HL),03H
100 7FC6 3E20            LD    A,20H
101 7FC8 189F            JR    AGAIN
102 7FCA 3ADB7F  TEST2:  LD    A,(LHSPCE)
103 7FCD B7              OR    A
104 7FCE 28E4            JR    Z,OUT4
105 7FD0 3D              DEC   A
106 7FD1 32DB7F          LD    (LHSPCE),A
107 7FD4 3E20            LD    A,20H
108 7FD6 1891            JR    AGAIN
109 7FD8 E1      OUT5:   POP   HL
110 7FD9 F1              POP   AF
111 7FDA C9              RET
112 7FDB 03      LHSPCE: DB    03H
113 7FDC 42      LNCNT:  DB    42H
114 7FDD 00      ENDPAG: DB    00H
115 7FDE 48      CHRLN:  DB    48H
116 7FDF E5      TOPPAG: PUSH  HL
117 7FE0 21DC7F          LD    HL,LNCNT
118 7FE3 3642            LD    (HL),42H
119 7FE5 E1              POP   HL
120 7FE6 C9              RET
121 7FE7 F5      DELAY:  PUSH  AF
122 7FE8 3AF67F          LD    A,(KBFIX+4)
123 7FEB EE10            XOR   10H
124 7FED 32F67F          LD    (KBFIX+4),A
125 7FF0 F1              POP   AF
126 7FF1 C9              RET
127                      ORG   7FF2H
128 7FF2 F5      KBFIX:  PUSH  AF
129 7FF3 C5              PUSH  BC
130 7FF4 012000          LD    BC,0020H
131 7FF7 CD6000          CALL  0060H
132 7FFA C1              POP   BC
133 7FFB F1              POP   AF
134 7FFC C35804          JP    0458H
135                      END

SYMBOL TABLE

AGAIN   7F69  BLNK    7FC1  CHRSLN  4029  CHRLN   7FDE
DLY     7F79  DLY1    7F7C  DELAY   7FE7  ENDLN   7F41
ENDPAG  7FDD  KBFIX   7FF2  LNPGE   4028  LPRT    7F34
LHSPCE  7FDB  LNCNT   7FDC  OUT1    7F8C  OUT2    7F91
OUT3    7FA2  OUT4    7FB4  OUT5    7FD8  RELOAD  7F4A
RUN     7F64  ROUND   7F82  START   7F66  TEST1   7FA8
TEST2   7FCA  TOPPAG  7FDF
```

ROM when the machine is switched-on.

The remaining two loadings are luxury items. When a disc system is not in use — several commands, such as FIELD, PUT, GET etc., will return "L3 ERROR" if you use them in Basic. According to the manual, that is because they are Disc Basic commands, which can be used only with a disc system.

In fact, each one of those commands causes the ROM to look in RAM for the address of a routine, and if you poke in your own address and your own routine, you can make use of any of the commands — there are 28 of them from which to choose. In the program, I have used the FIELD command to re-set the printer program to top-of-page, so that it will print a full 66 lines before putting in its six blank lines.

Delay times

The address of a page-re-set routine at 7FDFH is, therefore, loaded into 417DH, the address for the FIELD command. Similarly, I have used PUT to change the delay time, by loading the address of a delay-byte insertion routine to 4183H. By typing PUT and entering from Basic, the TRS-80 screen-access speed is considerably slowed so that all operations which involve the screen are run at low speed and that obviously includes listing.

If the TRON command, part of standard TRS-80 Basic, has been entered, so that the line numbers are printed as the program executes, PUT allows me to watch a program executing in slow motion — a most useful debugging aid for elusive faults. Typing PUT for a second time and entering restores normal running speed.

That delay, incidentally, can also be implemented by using the keyboard address, but the screen routine is preferable, because the delay is then operative only when the screen is accessed — the keyboard is scanned continually, and any delay in the keyboard routine slows the computing rate considerably unless the keyboard is disabled during computing time. By this time, re-locatability was abandoned.

The loading section ends with a jump to 0072H, which returns to Basic. That is not the address given in the older Radio Shack manuals, incidentally. The address in the old manuals is not so suitable, and can cause error messages to appear; later editions give this correct address of 0072H.

The print routine starts at 7F34H, with a DI command so that the routine cannot be interrupted in mid-character by the break or any other key. The printer routines of the TRS-80 place the character byte in the C register, so that is transferred to the accumulator, A, for subsequent operations. The section of program from 7F36H to 7F40H saves registers, and calls the screen-print routine, so that whatever is printed is also echoed on the screen.

That is a useful feature, since most dot-matrix printers conceal what is being printed. Curiously enough, that, along with the delay, causes odd effects — a "z" is printed in place of the least significant digit of the line number — when that routine is used along with Zen, so that the version of the program which I have tacked into Zen has 7F3AH and 7F3BH both set at 00, forming a set of NOP instructions.

At ENDLN, the byte in the accumulator is checked to see if it is a carriage return (0DH), in which case the program branches to RELOAD so as to re-set the memory location which governs the number of characters per line and to decrement the number-of-lines count in location LNCNT.

If at ENDLN, the byte in the accumulator has a value less than 20H and is not a carriage return, the program returns for the next character, since the other control characters are not used in this program. For any other character, the program jumps to START.

The registers are saved, and register B is loaded with a bit count — one byte plus a space. A mark bit is sent out by loading FC01 to HL and calling 0221H, which sends the bit in the L register to the cassette port. Incidentally, I have used 01 for mark and 10 for space, because this gives a much larger output signal than the conventional 01 and 00.

To establish the correct baud rate, a delay must follow, and that is achieved in the subroutine labelled DLY. The byte 00DEH which is loaded into HL for the delay subroutine determines the baud rate, so that if you are writing for 110 baud, a larger number must be loaded — try 0267H.

Carry position

At the label ROUND, the lowest bit of the byte is rotated into the carry position by the RRA command, and the other bits are similarly shifted in the accumulator. If the bit in the carry position is 0, C is re-set, and the program jumps to OUT1 to send a space bit. If the bit in the carry position is a 1, hl is loaded with FC02H, the space bit which is output by the routine at OUT2. Those routines are followed by the DLY subroutine to keep the baud rate correct.

The DJNZ command at 7F97H keeps the rotate-and-output routine going until the B-register is decremented to zero in the usual action of the DJNZ command. A final delay subroutine follows, and then at OUT3, which is the end of the main section of the print routine, the original unrotated byte is recovered and compared to the carriage return byte, 0DH. If the byte was a C/R, the program skips to TEST2, otherwise at TEST1 the byte from ENDPAG is loaded, and the accumulator is cleared.

Now if the lines-per-page count reached zero earlier in the program, at RELOAD, there will be a six in ENDPAG, so that the OR(HL) step will give a non-zero answer, and the program will continue to decrement ENDPAG, load in a carriage-return character and run it through the print routine again until ENPAG is zeroed.

If the line/page count has not reached zero, the step at 7FADH directs the program to BLNK, where the store LHSPCE is loaded in with the number 3. The accumulator is loaded with the ASCII blank character, 20H, and the printing routine repeated, so that a space is printed at the left-hand side of the page.

That continues until the store LHSPCE is decremented to zero, giving the left-hand margin which prevents the listing from being mangled.

When the character in A is not a carriage-return, the test routine is TEST2. That tests for the byte in LHSPCE to see if more spaces need to be printed, otherwise the retreat is sounded by jumping to OUT4. At OUT4, the characters/line count is decremented and, if the count is zero, the accumulator is loaded with a carriage return character and re-cycled. If the byte is not at the end of a line, the next stage in the countdown is OUT5, where the registers are restored and the routine returns for the next character.

Storage bytes

The bytes from 7FDBH to 7FDEH are storage bytes for the left-space, lines/page, end-of-page space, and characters/line respectively. At TOPPAG, a short routine restores the lines/page count. That is the routine which is called by typing FIELD and entering. Similarly at 7FE7H, a short routine inspects the delay byte at 7FF6H and XORs it with 10H. If the delay byte was 0, it is replaced by 10H, if it was 10H, it is replaced by 0, such being the action of the XOR command.

The routine is called from Basic by the PUT command. It, like FIELD, can be called from a running program, so that a program which prints to the screen can be slowed in mid-run and speeded up again as needed — useful for instructions. Finally, at 7FF2H, the video delay consists of saving registers, loading BC with the delay bytes, calling the TRS-80 delay subroutine at 0060H, restoring and then jumping to the screen routine at 0458H.

It has been a very useful routine for me, and its usefulness has been enhanced by a short program from A J Harding (Molimerx), which transforms the machine code of a routine into DATA lines of a Basic program. In that way, I can poke it in from any Basic program, so that I do not need to load a separate object code tape.

Furthermore, if I ever need it to be fully re-locatable, I can place the DLY subroutine and the store bytes into low memory between 405C and 407F — that is used only when the TRS-80 is first switched-on. At present, however, the ease with which I can generate object code with Zen makes strict re-locatability unnecessary.

SOMEONE, ON ONE OF THE INTERWEB CHANNELS WAS ENQUIRING ABOUT THIS COMPETITION - IT'S PART OF THE MOLIMERX-LTD CATALOGUE (WHICH IS AVAILABLE FROM THE DOWNLOADS PAGE AT TRS-80.ORG.UK)

Software News

INNOVATIVE
TRS 80-GENIE SOFTWARE

from the professionals

£250 REWARD

Below you will find described a new program entitled Enigma. It is a true simulation of the German wartime cypher machine of that name. It will encipher messages which may be communicated to third parties by any means who, assuming they have the key, will be able to use their Enigma program to decipher.

We will pay the sum of £250 to anyone [who has purchased the program] who can demonstrate an infallible method of deciphering the coded message supplied in the program's instructions. We consider Enigma to be the best program of its kind on the U.K. or U.S. market; contestants may therefore use any orthodox means to crack the code, including microcomputer programs other than Enigma.

The original message and keys will be lodged with our Solicitors for safe keeping in a sealed envelope. In the [hopefully] unlikely event that the code is cracked by more than one person, the reward will be paid to the first customer who demonstrates to us that he has succeeded.

MOLIMERX LTD.

During the 1939/45 war the German Army and Intelligence used a deciphering machine called Enigma. It was a fascinating machine and the stories that have surrounded it are equally interesting. There have been some four or five books written about the machine, and with regard to the way in which the British counter intelligence managed to crack the code.

That they did so was the culmination of some fortuitous circumstances, a lot of luck, but mainly it was due to the fact that the people who did it were extremely clever mathematicians. The fact that it took so much brain power, plus a rudimentary type of computer and a specimen of the machine in order to crack the code is an indication of how complex that code is.

The Enigma microcomputer program that we are selling is a simulation of the original machine, together with one or two improvements which were suggested by Gordon Welchman, who wrote the book "The Hut Six Story" last year and was also the leader of the team that cracked the code.

Although the machine and, therefore, the program is so complicated, its use is amazingly simple. One simply inputs a key and a message and the code is supplied. To decipher, the message is input again with the key and if the key is correct then the decoded message is displayed. With the cassette version it is necessary to input from the keyboard but with disks both inputs and outputs may be to disk files if required. A printer is of course supported.

The code may be transmitted in any way which the written word can be transmitted. Companies who wish to fully protect their communications will no doubt have the program generate the code and then tap it into a telex. Tape users will have to send either the output from their printer or write down the code direct from the screen.

Enigma is a fascinating program designed, not only for those people who are interested in encryption professionally or as a hobby, but also for companies or private persons who wish to communicate with others in an entirely secure manner. As is shown by the above Reward Notice, we have great faith in the powers of this piece of software.

ENIGMA (Tape) ... **£17.25**
ENIGMA (Disk) ... **£23.00**
Inclusive of V.A.T. P&P 75p

TEL: [0424] 220391/223636

MOLIMERX™ LTD
A J HARDING (MOLIMERX)
1 BUCKHURST ROAD, TOWN HALL SQUARE, BEXHILL-ON-SEA, EAST SUSSEX.

TELEX 86736 SOTEX G

TRS-80 & VIDEO GENIE SOFTWARE CATALOGUE £1.00 plus £1 postage.

GEORGE PHILLIPS

ANOTHER LITTLE PROGRAM TO PUBLISH IF YOU LIKE. I SAW A POSTING OF THIS CAT ANIMATION ON A COCO AND THEN AN MC-10. I THINK THAT IT HAS BEEN PORTED TO MANY DIFFERENT 8 BIT COMPUTERS. I FIGURED IT NEEDED A TRS-80 MODEL 1/3 VERSION.

=-- GEORGE --=

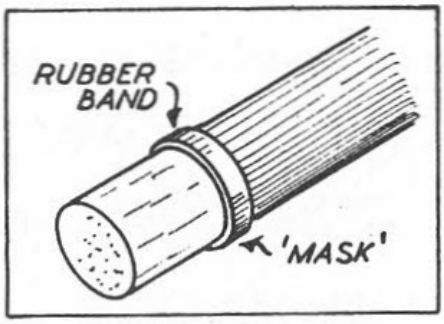
```
10 CLEAR1000:DEFINTA-Z:DIM C$(5,3):CLS
20 FORF=0TO5:FORY=0TO3:C$(F,Y)=" ":FORX=1TO16:READN:C$(F,Y)=C$(F,Y)
+CHR$(N):NEXT:PRINTC$(F,Y):NEXT:NEXT
30 CLS:F=0:P=-18:PRINT@640,STRING$(64,131)
40
IFP<0THENN=18+P:PRINT@384,RIGHT$(C$(F,0),N);:PRINT@448,RIGHT$(C$(F,1),N)
;:PRINT@512,RIGHT$(C$(F,2),N);:PRINT@576,RIGHT$(C$(F,3),N);:GOTO70
50 IFP>47THENN=64-
P:PRINT@P+384,LEFT$(C$(F,0),N);:PRINT@P+448,LEFT$(C$(F,1),N);:PRINT@P+512,
LEFT$(C$(F,2),N);:PRINT@P+576,LEFT$(C$(F,3),N);:GOTO70
60
PRINT@P+384,C$(F,0);:PRINT@P+448,C$(F,1);:PRINT@P+512,C$(F,2);:PRINT@P+5
76,C$(F,3);
70 F=F+1:IF F=6THENF=0
80 P=P+2:IF P>62 THEN P=-18:FORI=0TO200:NEXT
90 GOTO40
1000 DATA 128,128,128,128,128,160,176,176,176,176,176,140,140,140,164,148
1010 DATA 128,128,160,152,131,129,128,128,128,128,128,168,128,156,164,154
1020 DATA 134,131,183,176,152,140,140,131,131,131,131,131,137,143,156,144
1030 DATA 128,134,181,128,128,128,128,128,128,128,128,128,128,128,130,130
1040 DATA 128,128,128,128,128,176,176,176,144,128,160,176,176,128,160,128
1050 DATA 128,128,160,152,131,128,128,128,130,131,129,160,160,179,131,164
1060 DATA 131,163,185,165,176,140,134,131,131,131,131,175,139,172,179,129
1070 DATA 128,140,129,128,128,128,128,128,128,128,128,138,180,128,128,131
1080 DATA 128,128,128,128,160,152,134,131,131,140,176,176,144,176,180,128
1090 DATA 128,128,128,152,131,148,128,160,164,144,128,128,130,152,176,153
1100 DATA 130,131,131,128,138,152,140,129,128,162,155,139,186,144,128,128
1110 DATA 128,128,128,128,128,134,128,128,128,138,144,128,128,131,164,128
1120 DATA 128,128,128,176,176,176,134,131,131,140,176,128,128,128,144,128
1130 DATA 176,152,134,128,128,138,144,164,128,160,128,137,140,134,137,144
1140 DATA 128,128,128,128,128,128,130,190,169,165,156,140,140,131,131,128
1150 DATA 128,128,128,128,128,128,128,130,135,130,181,128,128,128,128,128
1160 DATA 160,140,140,140,176,176,140,140,164,144,128,128,128,144,128,128
1170 DATA 130,128,128,128,128,150,128,128,128,130,137,140,131,137,144,128
1180 DATA 128,128,128,128,128,169,152,140,140,173,164,140,131,131,128,128
1190 DATA 128,128,128,128,128,174,154,176,136,129,129,128,128,128,128,128
1200 DATA 130,131,131,140,140,164,176,140,140,176,128,128,128,160,184,144
1210 DATA 128,128,128,128,168,129,128,128,128,128,131,137,131,161,176,154
1220 DATA 128,128,128,160,134,176,152,131,137,140,140,140,172,185,128,128
1230 DATA 128,128,128,130,166,180,144,128,128,128,128,128,128,128,128,128
```

IF YOU WOULD LIKE A COPY OF THE BASIC PROGRAM, JUST TO SAVE HAVING TO TYPE IT IN, PLEASE EMAIL ME AND I'LL SENT ONE TO YOU ASAP. THERE WILL BE AN ANIMATED .GIF RUNNING ON THE WEBSITE (WWW.TRS-80.ORG.UK) SHORTLY. *ED.*

HANDY HINTS

HERE'S SOMETHING JUST A BIT DIFFERENT. IT'S A 'HANDY HINT' FROM THE 1955 'GADGETS ANNUAL'. I MUST ADMIT, I HADN'T THOUGHT OF THIS ONE MYSELF. I HOPE YOU FIND THE TIP USEFUL.

RUBBER BANDS AS
PAINTING MASKS

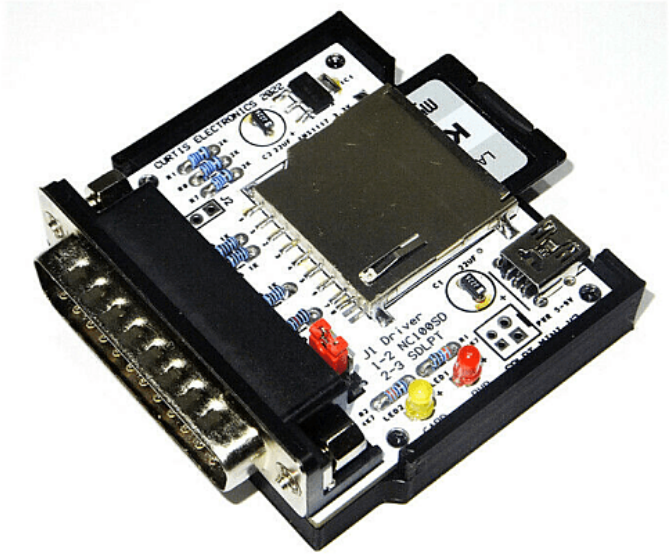


WHEN painting small articles, a flat rubber band will make a good mask to produce a clean, straight edge to the paintwork. Just slip the band in place up against the line marking the end of the paint and then paint right up to the band. Remove the band when the paint is still slightly tacky.

A tip when using two or more colours: paint the lightest colour first, allow to dry, and then apply the next lightest colour, and so on.

AVAILABLE ON THE U.K. EBAY SITE, I THOUGHT THIS ADVERT MIGHT INTEREST THE 'TANDY PC' COMMUNITY. I RECALL A CD DRIVE USING THE PARALLEL PORT IN THE EARLY 1990'S *ED.*

SDLPT SD-Card Adapter
LPT Parallel Port Removable SDCard Drive for Classic PC's



Read/Write Files Through LPT Port
Floppy Drive Transfer Speeds
Driver Supports DOS and Windows 3.1/95

THE SDLPT ADAPTER IS DESIGNED TO ALLOW CLASSIC IBM COMPATIBLE PC SYSTEMS TO USE SD-CARDS AS A DISK FILE STORAGE DEVICE. THE ADAPTER IS AN ALTERNATIVE OPTION TO FLOPPY DISCS AND IS NOT DESIGNED AS A DIRECT REPLACEMENT FOR HARD DRIVES OR OTHER STORAGE DEVICES.

THE DRIVER USES A SLOW SPI (SERIAL PERIPHERAL INTERFACE) CONNECTION THROUGH THE LPT PORT TO COMMUNICATE WITH THE SD-CARD AND THIS LIMITS THE DATA TRANSFER RATE. TYPICAL FILE TRANSFER RATE IS COMPARABLE TO A FLOPPY DISK DRIVE AT AROUND 30-60 KB/S. TRANSFER RATES ALSO DEPEND ON THE PC SYSTEM CONFIGURATION AND LPT PORT SETTINGS.

THE ADAPTER IS COMPACT ENOUGH TO NOT PROTRUDE TOO MUCH WHEN PLUGGED INTO THE LPT PORT.
LOCKING NUTS CAN BE USED TO SECURE THE ADAPTER IN PLACE OR A CABLE CAN ALSO BE USED INSTEAD
OF DIRECTLY PLUGGED INTO THE PORT.

POWER IS SOURCED FROM A 5V MINI USB PLUG OR OTHER 9V TO 5V 30MA POWER SOURCE.
THE ADAPTER CAN ALSO BE POWERED FROM A 9V PP3 BATTERY. MINIMUM 5V REQUIRED.

EBAY.CO.UK 'BARGAINS'

HERE ARE JUST SOME OF THE MORE INTERESTING TANDY EBAY LISTINGS, SINCE OUR LAST ISSUE. WERE YOU ONE OF LUCKY ONES? HOW ABOUT LETTING US ALL KNOW HOW YOU'RE GETTING ON?

£
£20

£99

£300

£37

£110

£311

£210

£105

£230

EBAY.CO.UK 'BARGAINS'

HERE ARE JUST SOME OF THE MORE INTERESTING TANDY EBAY LISTINGS, SINCE OUR LAST ISSUE. WERE YOU ONE OF LUCKY ONES? HOW ABOUT LETTING US ALL KNOW HOW YOU'RE GETTING ON?

£

£78

£205

£62

£135

£576

£51

£112

£190

£4

Retrocomputing Pages: TRS-80 Model 100

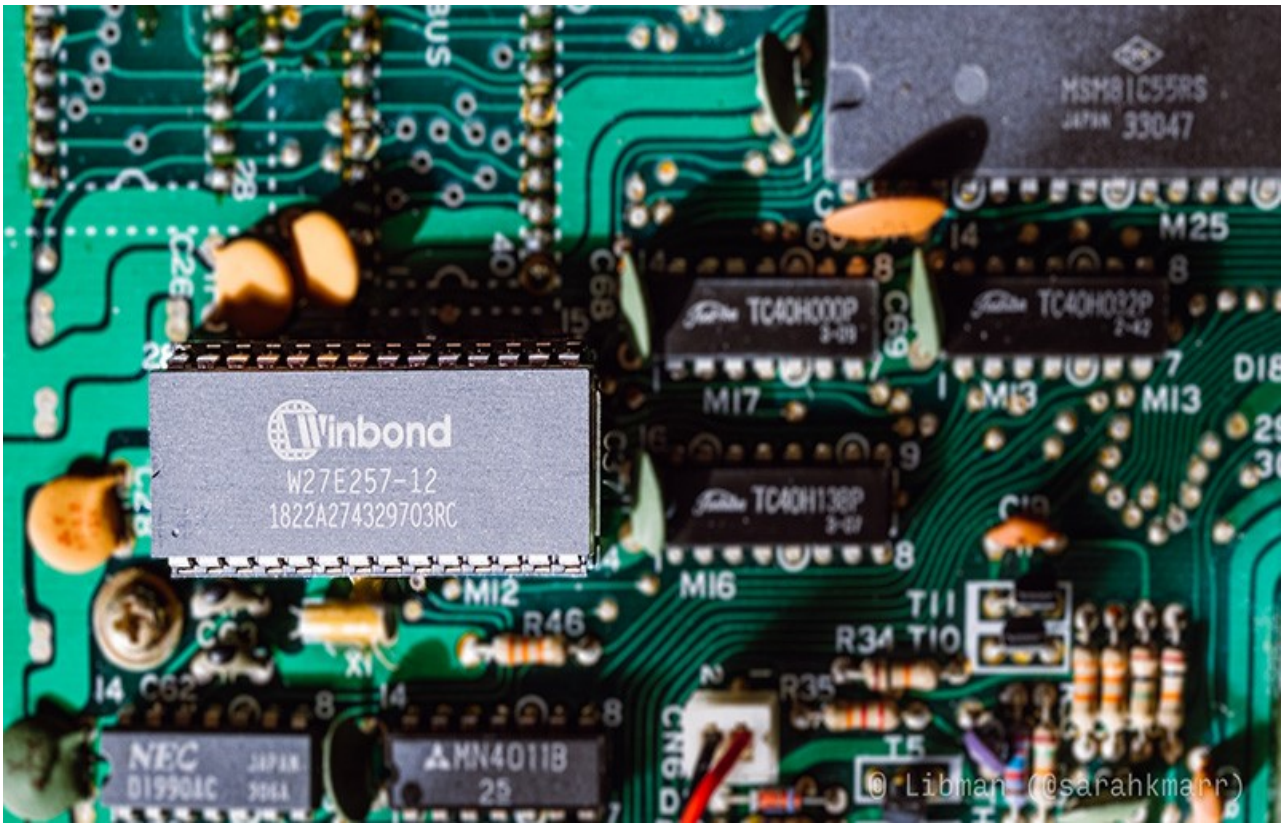
Sarah Libman

Quick download link: LibROM

When I got my TRS-80 Model 100, a UK model, I discovered that UK Model 100s don't have a modem. Eh, no big deal. But! what was a big deal is that UK ROMs have all their routines in different locations to American ROMS, so nothing which uses ROM calls will run on them; REXCPM isn't compatible, etc. Anyway, fast forward and I've learned about the Model 100, picked up 8085 assembly, got to work on Ken Pettit's annotated ROM listing, disassembled Ron Wiesen's 'Teeny' — with helpful pointers from Steve Adolph — and hex-edited my way to LibROM.

Click here to download the zip file, which contains a .bin file to be program an EPROM and a .asm file which contains a heavily annotated assembly listing of the ROM. Although LibROM was designed for non-American machines, it should work in American Model 100s, too. (Note, however, that early American models used non-standard pins for the ROM chip so it may not be trivial to replace it. Also, you might want to back out the change of date format, for American use.)

I used a XGecu TL866II Plus EPROM programmer, which did the job without any problems. I programmed the ROM onto a Winbond W27E257 EEPROM, which works fine in the machine, and which has the advantage of being electronically erasable and reprogrammable.



I haven't had time to test the ROM as thoroughly as I'd like. Everything seems fine, but I wouldn't use it for mission-critical Model 100 operations. It works in VirtualT, although there seems to be a VirtualT bug which prevents Teeny+ for loading large files: VirtualT doesn't refill the loading buffer once the first filled buffer has been emptied. (I have loaded large files on the actual Model 100 using Teeny+.)

Description

LibROM started as an adaptation of the Model 100 American ROM for UK (and other non-American) machines which lack a modem, so that software and external ROMs which make calls to ROM routines will work. It grew from that to include other bug fixes and enhancements - including in-ROM disk management - replacing modem, address and schedule code to do so.

BASIC start screen
showing: ROM details & UK date formatting

MENU screen
showing: status line (with time, date, day, sound icon, current STAT settings & reassuring message) & TEENY+ item

TELCOM screen
showing: modem settings interpreted as 300 baud RS-232 settings, & file list & free space commands

TEENY+ screens
showing: load, kill, save & check functions; short filenames, auto-capitalization, OK messages; STAT toggling; DSR toggling; free space and files commands

Communications

- Removal of modem functions: LibROM always sets ports to use RS-232 and not the modem. If a STAT is entered beginning with 'M' it is interpreted and stored as the same stat for RS-232 with a baud rate of 300. For example, "STAT M8N1E" results in a STAT of 38N1E. The CALL (i.e. dialing) function has been removed.
- The ADDRSS and SCHEDL functions are removed, to make room for disk functionality.

Changes to TELCOM: the "Find" (F1) and "Call" (F2) instructions are replaced with "File" (F1), to display a list of files in memory, and "Free" (F2), to display free memory.

Localization

DATE$ processes dates in UK (non-American) DD/MM/YY format (as described in the UK manual), and the MENU screen shows the date in that format. (The storage format of the date in memory is unaffected.)

Display

- The MENU screen top line now shows: time, date, day, a sound on/off icon, the current STAT settings and a reassuring message
- Character 88H (136) is changed from a sans serif 'i' to 'π'. This matches the (incorrect) UK version of the "Model 100 Quick Reference Guide" and the Model 102 character set, and seems more useful.
- The text for July is changed from "JLY" to the more usual (and Model 102's) "JUL".

The capitalization of "Bytes free" is standardized to "bytes free".

Sound

Global SOUND ON / SOUND OFF: these functions now turn all sound on/off, not just cassette and modem sounds. This includes system beeps. Run times are unaffected: a 2 minute tune with sound on will be two minutes of silence with sound off. The MENU-screen sound icon is changed accordingly.

The PRINT Button (Sound and Scrolling)

- The PRINT button is repurposed to toggle sound on and off. This does not need to be done when on the MENU screen. The MENU-screen sound icon is changed accordingly.

Shift+PRINT toggles now toggles scrolling on and off. This does not need to be done when on the MENU screen. Note that many applications, including the MENU screen, also turn scrolling on or off, so settings may change without use of shift+PRINT.

Bug Fixes

- The ATN function is fixed to use correct (Model 102) coefficient in calculations.

The "search for file" ROM routine is fixed to allow searching for 6+2-character filenames, not just 4+2.

Disk Management: Teeny+

Ron Wiesen's Teeny disk management software is implemented in ROM and augmented to create Teeny+. An entry for Teeny+ is added to the MENU screen. Teeny+ makes the following changes to Teeny:

- the command memory is cleared before each entry to prevent unwanted results

- there is no longer a need to add spaces to filenames shorter than 6+2 characters

- (e.g. the file HI.BA can be loaded using "L HI.BA" rather than "L HIssss.BA", where 's' is a space)

- commands are automatically capitalized so a mixture of lower- and upper-case characters can be used on entry (e.g. "S fred.do", "s FRED.DO", "S fReD.dO" are all valid and all execute "S FRED.DO")

- there is no longer a need to supply a dummy filename for commands which do not require a filename

- Teeny+ sets the communications STAT to 98N1D when executed, and displays it to the user, but saves the existing system STAT beforehand, so that other STAT values can be used in Teeny+, if required
- the displayed STAT value has a suffix of 'd' if a DSR check is required, and 'x' if not
- the 'B' command toggles between 98N1D and the saved system STAT, and displays the newly active STAT

- exiting Teeny+ restores the system to the saved system STAT (rather than leaving as 98N1D)

- the requirement for a DSR check can be toggled with the 'D' command; the new setting is displayed

- commands have been added to list files in memory ('F') and display empty (free) space in memory ('E')

- the prompt text has been updated to reflect changes from Teeny

- the error message text has been updated to report "msg" rather than "Err" as some messages are not errors

- the K, L & S functions now report an "OK" message when executed successfully - the command to exit Teeny+ is now 'M' (for "MENU")
the command to check for the existence of a file on disk is now 'C' (for "Check")
(In fact, just like Teeny, Teeny+ interprets any command which is not K, L, S, B, D, E, F or M as "check file on disk".)

Full list of commands:

C file - check if file exists on drive
K file - kill (delete) file on drive
L file - load file from drive to local memory
S file - save file from local memory to drive
B - toggle baud and other settings (STAT) between saved system STAT and 98N1D
D - toggle requirement for DSR on ('d') and off ('x')
E - display empty (free) space in local memory
F - display list of files in local memory
M - return to MENU (exit Teeny+)

2022 Competition Wrap-Up
George Phillips

Half of last year's competition was to demonstrate the best compression of the Wordle dictionary and answer list. The programmers who entered have all been gracious enough to share their source code. I've taken some time looking over their code and will talk here a little bit about general ideas in compression and give my understanding as to the theory of operation of each entry.

Before we continue I must apologize to Don Barber who submitted a very good 6809 entry which slipped between the cracks and didn't get reported in the results. He was able to get the size down to 21664 bytes which put him solidly in 4th place. Sorry about that, Don, and thanks for taking the time to enter.

The Wordle dictionary (as was set for the competition) is 12,960 five letter words. A subset of 2,313 of those are specially designated as answers. The game will pick one of those words for the player to guess. To be true to the original game the answers must be remembered in order. Roughly speaking, the game would look at the number of days that have passed since it was released and use that as the answer number to present. This meant that every player would be guessing the same answer on a given day and would get a new puzzle every day.

As you might imagine, it is easier to compress a list of words if the order doesn't matter. There's simply less information to record. One way to approach things is to compress the entire 12,960 word dictionary in whatever order is most favorable. Then represent the answers as an ordered list of the dictionary entries by number. Or you could compress the 10,647 valid-but-not-answer words as best you can and treat the remaining list of 2,313 answer words as a separate problem. In either case the unordered list of words is much bigger and easier to compress and thus a good place to focus our effort. We have the most to gain and the best chance of realizing those gains.

Let's start with the full 12,960 word dictionary. If you look at a text file containing one word one per line it'll be 77,760 or 90,720 bytes in size. You'd think that it should be 64,800 bytes because there are 12,960 words at 5 letters per word. The extra bytes come in due to the characters used to indicate the end of each line. Due to a long standing tradition of incompatibility we in the computing world have never come to a universal agreement of how to denote the end of a line. Windows and their brethren use 2 characters, unixy computers use 1. Despite the disagreement the extra data is necessary for a text file as the lines can vary in length.

In our case we know every word (and thus every line) is 5 letters. We can drop all the line endings and still easily be able to read out the word list. Well, a computer can. I'm not so sure I can quickly rattle off the words in “polypfinalmushyreplygrain”.

While it was a fairly simple observation it was nonetheless a big saving dropping at least 12,960 bytes and getting us down to the expected 64,800. Or at least in the ballpark. After all, we still need to remember how many words are in the list and there is still the code access it.

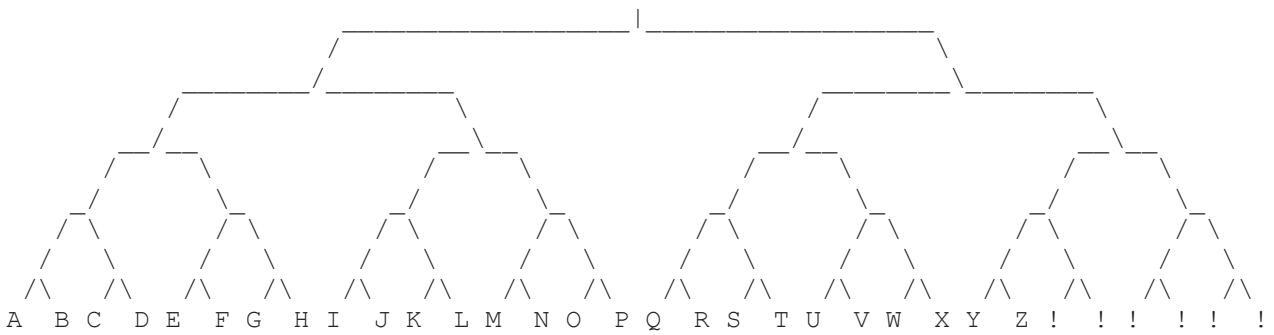
For our next step let's ask if one byte per letter is the best we can do. Considering that a single byte can store 256 (2^8) values and we only have 26 possible letters it sure sounds like a lot of wasted space. With a little trial and error we can figure out that 5 bits is enough. A single bit can store 2 values, 2 bits can store 4 values and so on. In general, n bits can store 2^n values. So it goes that 5 bits can store $2^5 = 32$ values. Mathematically we can get the result directly by calculating $\lceil \log_2(26) \rceil$. Those broken square braces mean “the ceiling” which is a fancy way of saying “round up”. If your calculator doesn't have base 2 logarithms you can replace $\log_2(n)$ with $\log(n)/\log(2)$. $\log_2(26)$ is just a smidge over 4.7 which rounds up to 5.

At 5 bits per letter we're using a mere 25 bits per 5 letter word. That's much better than the 40 bits at 1 letter per byte we were using. We could do the math to figure out how big the compressed data will but it's just a well to keep thinking in terms of bits per word. As long as we think the extra code won't be bigger than the savings we're OK. For 12,960 words even saving 1 bit per word is 1,620 bytes which is an awful lot of assembly code.

Stepping through memory 5 bits at a time is more complicated than going a byte at a time. And certainly it's slower but speed isn't a problem in this competition so never mind that. The Z-80 code required is only about 20 or 30 bytes. Take a look at Mathew Boytim's entry for an example. `shiftb` is the subroutine that pulls `B` bits into `E` register. And just after `fwini:` you'll find the code to initialize the bit reading.

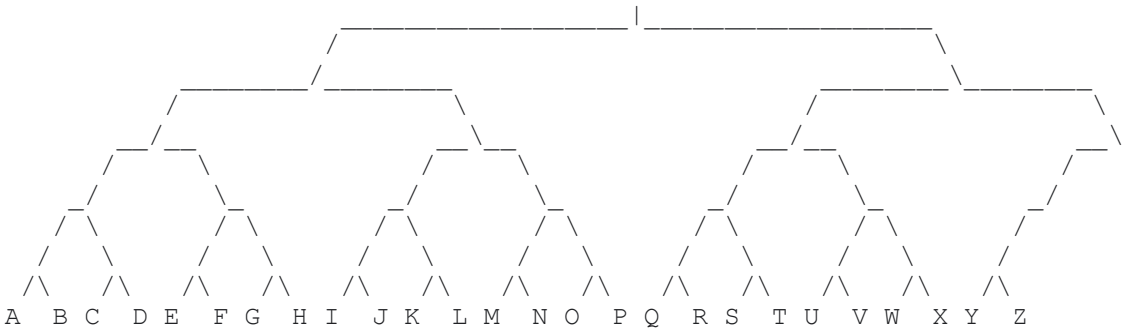
Can we do improve on this? Well, it is clear that 5 bits is more than we need. Only 26 of 32 possible values are used. One idea would be to use the extra 6 values to mean something special. For example, maybe make 27 correspond to some common two letter sequence like “th” or whatever the most common letter pair that occurs in the list. That'll be more assembly code, of course, but feels like we'd come out ahead. For that matter, maybe we should go back and look at those byte values. They wasted a whole lot more but with a bit of thought we could find good uses for the 224 unused codes. It is a fruitful approach and the code will be faster and by extension smaller working a byte at a time. See Kyle Wadsten's entry for a good example of adding special codes.

For now let's confine ourselves to better packing. Instead of thinking each letter as a 5 bit chunk let's lay the codes out in a tree. The idea here is that we start at the root of the tree and read a bit. If the bit is 0 we go left otherwise we go right. That would look like this:

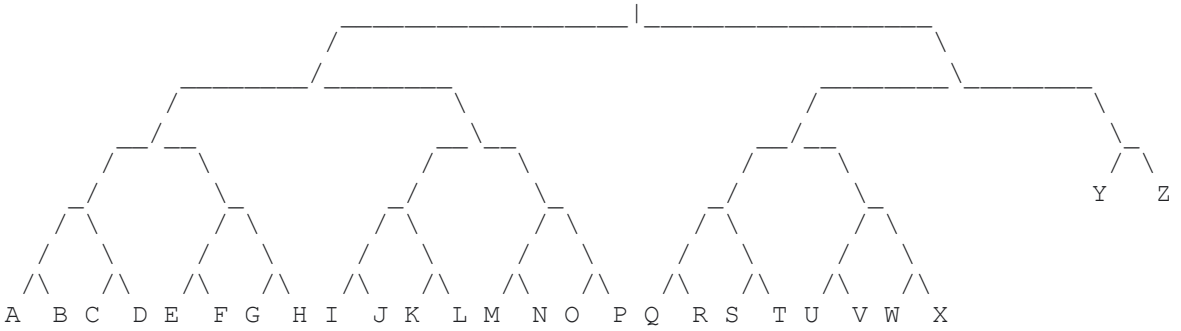


If we see 5 zero bits in a row (00000) we got all the way down the left side of the tree and end up at 'A'. If the next 5 bits are 10101 we wind our way down to 'V'. Going too far to the right of the tree gets us into the unused area. Pure wasted space.

Let's chop out those unused paths to make our tree neat and tidy:



The actual waste now becomes clear. As we head to either 'Y' or 'Z' we hit two nodes where there is no choice as we always turn left. We may as well just move 'Y' and 'Z' up two levels:



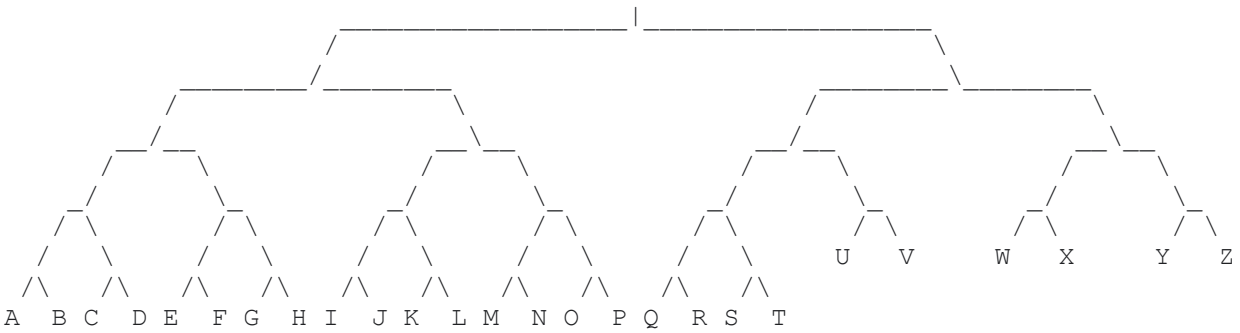
Nice picture, but so what? Well, if we follow our rule we find that 'A' through 'X' still have 5 bit codes but 'Y' and 'Z' have three bit codes: 110 and 111. It feels like this new tree should work but does it really? We can no longer just read 5 bits but need to decide, somehow, if we should read 5 bits or 3. That could be a lot of code or control data.

Looking carefully at the paths it will become apparent a simple rule will suffice. Notice that the letters 'A' .. 'H' start with two zero bits. 'I' .. 'P' start with 01 and 'Q' .. 'X' start with 10. Meanwhile, 'X' and 'Y' start with 11. All we need to do is:

> read 2 bits
> if they're 11 then read one more bit to pick 'Y' or 'Z'
> otherwise, read 3 more bits and use the last 4 bits to choose the 16 letters 'A' .. 'X'.

Will this be worthwhile? There will definitely be less than 5 bits/character. 24 of the letters still have 5 bit codes but 2 of them are 3 bits long. On average there is (24 * 5 + 2 * 3) / 26 bits per letter which is about 4.846 bits/character. 0.154 bits/character saved doesn't sound like a whole lot but it works out to 1,247 bytes.

In fact, we can do a little bit better by balancing the tree. Push 'Y' and 'Z' down so that 'U', 'V', 'W', and 'X' can have shorter codes.



Now we average (20 * 6 + 6 * 4) / 26 = 4.769 bits/letter. The improvement is slimmer but I still think even with a bit of extra code we'll come out ahead.

In either case there's a more important conceptual breakthrough is we've moved from using an integer number of bits per character to a fractional number. It's easy to get stuck on the idea that the smallest unit in the computer is the bit. Not true! Here we are with fractions of a bit.

It really gets you thinking about how we got to 5 bits in the first place by rounding up $\log_2(26)$ to 5 from its actual value of about 4.70044. Maybe that wasn't just a step in figuring out how many bits we needed. Maybe that is *exactly* the number of bits we need. I hate to gloss over the whole discipline of Information Theory but in short, yes, it really is! This is good news; we can do better than 4.769 bits/letter. The bad news (and you knew there was bad news) is that the theory isn't telling us how to get that value.

As it turns out, base 26 encoding of the all the words will get to the 4.7 “minimal” bits per letter predicted. In my example program I encoded each 5 letter word as a base 26 number. HAZEL has letter numbers 7, 0, 25, 4 and 11. As a base 26 number it is $7*26^4 + 0*26^3 + 25*26^2 + 4*26^1 + 11*26^0$ or 3,215,847. The largest possible word Is ZZZZZ which is 26^5-1 or 11,881,375. $\log_2(11{,}881{,}375)$ is about 23.5 so treated as numbers each letter can fit in 24 bits. That works out to 4.8 bits/letter; we get close to the best but lose out because we rounded up to 24 bits.

To get to 4.7 bits/letter we can't stop at a single word. Instead, we must encode all the words as one big number. How big? Well, 12,960 words gives us 64,800 letters. The single number holding all the words will be less than $26^{64,800}$ and the number of bytes will be $\log_{256}(26^{64,800})$ which rounds up to 38,074 bytes. The bits per letter is 38,074*8/64,800 = 4.7005 which is pretty darn close to the theoretical minimum of 4.70044 bits/letter.

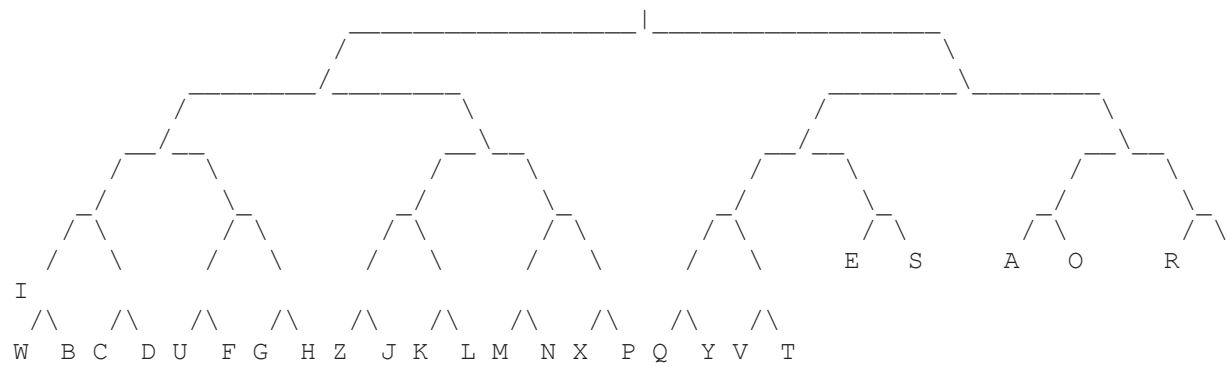
The theory looks good but the practice has a few problems. Every letter extracted requires dividing a 38K digit number by 26. The number will get smaller as we go but roughly speaking that's on the order of 38K * 38K operations. Even if each division step were a single instruction that's an awfully long time to search through the dictionary. Even worse, we need to keep track of the quotient after each division. We'll need 38K of memory for the dictionary and 38K working space as we extract the letters.

That won't fit on a Model 1 and might barely work on a Model 4. At best it might be plausible on the Model 16's 68000.

There may be some sort of middle ground. Base 26 encoding 5 letters at a time wasn't as efficient as the packed tree method but maybe 100 letters at a time would get there without being as expensive. Still, I think it best at this point to consider this a dead end and look to see how the tree method can be improved.

Right away there is a small problem. I calculated the bits/letter as an average. That's fair enough for an approximation but the dictionary is fixed. There's no need to guess at the number of bits since it can be calculated exactly. I'll just whip up a little script to count 5 bits for 'A' .. 'T' and 4 bits for 'U' .. 'Z' (… a bit later ...) and it's 4.89 bits per letter. Uh-oh, that's worse than predicted!

Why is that? If I print out the number of times each letter appears it becomes clear – while 'U' and 'Y' are reasonably common, 'V', 'W', 'X' and 'Z' are some of the least common letters. Yeah, I know, this is hardly a revelation and I'll confess I glossed over the issue for the sake of the story. Most importantly, there wasn't even a need to guess the expected frequency of each letter when the data is known. And having written a little program to count letters it's really easy to show that the six most common letters are 'E', 'S', 'A', 'O', 'R' and 'I'. [Is it mere coincidence that Saoirse Ronan has the most common letters in her first name]. What if we arranged the tree so that those letters used the short 4 bit codes? Sounds like a real pain shuffling all the branches around so instead I'll just swap the leaves around, 'E' ↔ 'U', 'S' ↔ 'V' and so on:



Will this have much impact on the code side? Not really; at worst it will be a few bytes of code to look into a 26 byte table to remap the letters. By choosing the most common letters for the short codes the bits per letter drops down to 4.512. This is excellent but how can it be better than the 4.7 value from theory?

As is usually the case, the math is good but the assumptions have led us astray. That $\log_2(26)$ business assumed that each letter is equally probable which clearly isn't the case. And before we get too mad, we were mostly looking at situations where a letter or a word would be packed into some fixed size unit. That ease of indexing can have a lot of practical value and is nothing to be ashamed of but here we want the best compression.

Having found a loophole of sorts the next thing to ask is if we can exploit it more. The current tree structure was arrived at just to avoid wasted code values.

The occurrences of each letter vary quite a lot. 'E' and 'S' appear about 6600 times each where 'J' and 'X' are only 290 and 'Q' a paltry 112. Surely we could change the tree so that 'E' and 'S' have the shortest possible codes and letters like 'Q' and 'X' longer ones? Feels like we should come out ahead, right?

Yes, yes we can. This idea of building a tree where the most common letters get the shortest codes and uncommon ones get the longest is called Huffman Coding. Read about it here:

https://en.wikipedia.org/wiki/Huffman_coding

Building a Huffman tree based on the occurrences of each letter I got these codes:

```
S 000    G 00110   N 01111     Q 011111101
E 100    H 01110   V 010110    X 111111101
T 0010   P 01001   W 110110
L 1010   C 11001   F 011101
I 0001   M 11110   K 011111
R 0101   Y 01101   B 111111
O 0011   D 01011   Z 0111101
A 0111   U 11011   J 01111101
```

That works out to a pleasing 4.347 bits/letter. It amounts to a 1,336 byte saving over our previous best. We will pay some overhead due to more complex assembly code and some data to describe the tree. As you might imagine there are techniques to minimize the space required to store the tree; start here for some useful ideas:

https://en.wikipedia.org/wiki/Canonical_Huffman_code

Huffman coding is a very good technique and was used in many of the competition entries. It even promises to generate an optimal code for an alphabet given the input probabilities. Mind you, that's a story we've heard before. Are there hidden conditions on that optimality?

Indeed there are. Huffman codes are restricted to outputting a bit at a time. This means they can only match probabilities that are an inverse of a power of 2: $1/2^k$ for integer values of k >= 0. Unless the frequency of every letter is one of these values there will be some loss of compression efficiency. Information theory once again can tell us the minimum number of bits for encoding a letter depending on the probability, P, of that letter appearing. This is called the entropy of the letter and is given by $-\log_2(P)$.

We can take the probability of a letter appearing as the number of times is appears divided by the dictionary size. For example, 'E' and 'S' have a probability of about 6600/64800 or 0.109. If we had a way to encode at the entropy lower bond for each letter we'd reach 4.312 bits/letter. Not a huge difference but amounts to around a 280 byte saving.

The other inefficiency is how we compute the probability of a letter. The total number of occurrences divided by the number of letters in the dictionary is all well and fine but takes no account of context. If the letter 'Q' appears than the next letter is 'U' with probability nearly 1. An 'H' will be more likely than usual if the previous letter was 'T'. In general the idea is to create a model of the input that predicts the probability

of each letter coming next. It could take into account the previous letter or all the previous letters in the word or even the previous word.

That kind of idea can work well but there are two challenges. One is that each context will need its own Huffman tree – say 26 trees; one for each possible previous letter. The compression gains will be offset by the cost of storing all those trees. The other is the inefficiencies in the Huffman code. Suppose our predictions of the next letter get very good and we can confidently state the next letter with 0.75 probability. Ideally that would be encoded as $-\ln_2(0.75) = 0.415$ bits but the Huffman code will always use one bit. Our model is doing a great job but all its work is lost in the encoding.

Happily there is a coding technique that can get very close to the entropy of a letter. It has the slightly strange name of “Arithmetic Coding”. I find the theory of it fairly easy to understand but must admit to not yet being able to grasp how it is implemented. I leave the interested reader the Wikipedia article on it as a start:

https://en.wikipedia.org/wiki/Arithmetic_coding

Alan Petrofsky uses something like it in his competition entry: a particular type of Asymmetric Numeral System or ANS which has the coding efficiency of Arithmetic Coding with the speed of Huffman coding.

At this point we've reached the bleeding edge of data compression theory. The best data compression algorithms involving combing a model that predicts what data follows very well with a good entropy encoding like Arithmetic Coding. However, there are other very good compressors that work in entirely different ways. Often times on the simple notion of data literally repeating itself. Generically these are called LZ compressors and work by emitting codes that say “look back N bytes and repeat those K bytes”. They're very fast and have the advantage of dynamically adapting to the data as they read it. The Deflate compression in .zip files use this technique. They're not as well suited for our needs since a dictionary doesn't have much that is literally repeated and we know exactly what we're compressing up front. There's no need to adaptively compress when we can jump in with the best strategy immediately.

The point being that despite the length of this little journey into data compression I've only scratched the surface. And for every generic data compression that are many more suited to particular types of data that has its own patterns to exploit. There are so many ways to crack this nut.

To end this I'll mention one more approach as it comes up in so many entries. I'll refer to this as a delta list. Convert all the words into numbers using base 26 encoding and put them in ascending order (which will be the same as alphabetical order). Now write out the list as the numeric differences between each word. For the most part the differences will be small – the jump from HAZEL to HAZER is only 6. The biggest jumps will come when the first letter changes like from HYTHE to IAMBI but even those won't be much bigger than $26^4 = 456{,}976$. That fits in 19 bits giving us a decent 3.8 bits/letter. Now use some technique to write the common short differences with fewer bits and the efficiency will go up considerably. See the entries themselves for ideas on how to do so.

The Entries

Here's an overview of how each of the entries work. It's based on my look at the code so any mistakes are mine. Presented in no particular order. You can get all the source code at:

http://48k.ca/comp2022.zip

Don Barber

This was the only submission written in 6809 assembly. The answer list is stored separately from the rest of the words in a straightforward base 26 encoding so 3 bytes per word. The rest of the words are stored as a delta list with a twist. Before being alphabetized the letters of each word are permuted like so:

B C D E F

F E C D B

This has the effect of reducing the average size of the deltas and thus decreases the size of the compressed data. The numeric value of the first word is stored in the code which checks that for a match then proceeds to read a list of deltas. Each delta is a fixed-size 5 bit number giving the number of bits in the delta itself. A delta of 0 is used to terminate the list.

Mathew Boytim

All the words are stored in a single dictionary in alphabetical order as a delta list. The answer list is an array of 14 bit indices into the full dictionary. The deltas use intricate scheme that adds a delta value onto each letter individually. Letter 2 gets a one bit value added, letter 3 a 2 bit value, letter 4 a 3 bit value and letter 5 a 4 bit value. This amounts to a 1+2+3+4 = 10 bit delta value. But since the delta for letters 2, 3 and 4 are often zero there is an initial 2 bit value indicating the first letter with a non-zero delta. This leads to 4 possible deltas:

00 1-bit 2-bit 3-bit 4-bit – 4 values added to letters 2, 3, 4 and 5.
01 2-bit 3-bit 4-bit – 3 values added to letters 3, 4 and 5.
10 3-bit 4-bit – 2 values added to letters 4 and 5.
11 4-bit – 1 value added to letter 5

4 bits isn't enough to always reach the next 5th letter. And that might also be the case for the other letters. When that happens the first delta value in the list will be set to 0. The deltas are added as normal but the decoder knows this is an intermediate value and doesn't consider it a valid word. Instead it gets the next delta to move on to the next word.

The delta values are added onto in wraparound (aka modulo 26) fashion. Adding 3 to letter 24 ('Y') yields letter 2 ('B'). The delta for letter 2 is a bit different. Whenever it wraps around it adds 1 to the first letter. That's how the encoding avoids storing a delta for letter 1 which would be 0 most of the time and 1 only 26 times.

There's another special case where if the delta to letter 2 is 0 then 2 is added onto letter 2 with the carry propagating to letter 1. But the delta is still treated as a 0 and so the word generated is considering an intermediate word and the program moves on to the next delta.

Shawn Sijnstra

Here the entire dictionary is stored as a single delta list. The answers are stored as 14 bit indices into the dictionary. The deltas come in 6 different sizes: 3, 6, 9 12 and 16 bits. The deltas avoid redundancy by starting off at the maximum value for the smaller delta. A 3 bit delta will naturally indicate differences of 1 through 8. A difference of 0 is not needed at it would repeat the previous word. Thus the 6 bit delta starts at 9 and goes to 72 instead of a 0 to 63 range. The deltas are packed into 4 bit chunks (nybbles) and use 1 to 4 bits of the first nybble in denote the size of the delta.

The delta values themselves are moved around a bit as some longer deltas are more common than shorter ones. For instance, a skip of 3 is stored in a 6 bit delta with a skip of 10 taking its place in the 3 bit deltas. Here are the possible delta values and how they are encoded:

0xxx – adds 1, 2, 10, 4, 5, 6, 14, 15.

10xx xxxx – adds 9, 3, 11, 12, 13, 7, 8, 16 ... 22, 168, 24 ... 27, 86, 29 .. 41, 100, 99, 162, 45, ... 48, 104, 89, 51, 52, 130, 98, 90, 56 ... 58, 156, 60, 78, 88, 63, 64, 96, 66, 67, 97, 182, 103, 84.

110x xxxx xxxx – adds 73 ... 77, 61, 79 ... 83, 71, 85, 28, 87, 62, 50, 55, 91 ... 95, 65, 68, 54, 43, 42, 101, 102, 70, 49, 105 ... 129, 53, 131 ... 155, 59, 157 ... 161, 44, 163 ... 167, 23, 169 ... 181, 69 ... 584.

1110 xxxx xxxx xxxx – adds 585 plus the 12 bit value giving 585 .. 4680.

1111 1101 xxxx xxxx xxxx – adds 164,248 to the 12 bit value.

1111 xxxx xxxx xxxx xxxx – adds 70,217 to the 16 bit value.

It may look like there is quite a bit of data required for 3, 6 and 9 bit offsets but they are represented quite compactly by a list of values that swap out from the “natural” offsets.

There is one other special case to mark the end of the list:

1111 1100 – end of list

I did a fair bit of decoding to figure out the offsets so it is quite possible that I missed a case or have the offsets wrong. Look at the original code to be sure.

Bill Kewley

A Huffman code is used for the letters. The answer list is stored separately and encodes a full 5 letters per word. The rest of the words are stored as 26 lists of the 4 letter suffixes for each word. Each list starts out with a 2 byte count of the number of suffixes. And the lists are put in alphabetical order so there is no need to store the first letter. This makes for quite a savings as the first letter in an alphabetical list is highly predictable.

Kyle Wadsten

The answer list is stored separately as base 26 encoded words. The rest of the words are stored with an optimized base 26 packing. The words are broken into 3 letter prefixes and 2 letter suffixes. The prefixes are base 26 packed into two bytes with a maximum value of 17575 = 26^3 – 1 or $44A7 in hexadecimal. The suffixes come in two forms. One is a literal base 26 packing into two bytes which will have a maximum value of $2A3 in hexadecimal. Or it can be a single byte with the high bit set referencing one of the 128 common suffixes stored in a table. The dictionary must start with a prefix code but then the suffixes and prefixes may occur in any order. A suffix means we have a valid word with the current prefix and a prefix replaces the current prefix. Compression is gained because a common 3 letter prefix only has to occur once and common word endings are represented in a compact fashion.

In order to distinguish the prefix codes from the suffix codes the prefix codes have $4000 hex ORed to them. The codes are stored in big endian fashion (high byte first). The code can read a byte and know what it has based on the high two bits of the byte:

01 – a 3 byte prefix.
00 – a 2 byte suffix.
10 or 11 – a single byte common suffix; the bottom 7 bits are the table lookup.

There is one last wrinkle. The 3 letter prefixes can be as large as $44A7 so after the OR with $4000 you can't know if the prefix had that bit set or not. This is handled by putting the prefixes in alphabetical order and noting the address in the table where the first prefix >= $4000 occurs. The address of the prefix is checked and if it is below that transition address the code knows that the bit must be cleared otherwise it can be left alone.

Peter Phillips

The entire dictionary is stored as a delta list. Each delta in the list reports the number of bits required to represent the delta and whether or not the next word is in the answer list. The number of bits required is one less than the actual number since the top bit must necessarily be 1. A zero bit delta represents a 1. A one bit delta can either be 10 or 11 in binary or 2 or 3. A two bit delta covers numbers from binary 100 to 111 or 4 through 7. These bits are added to an answer bit and put into a Huffman tree to minimize the space required to represent them as deltas with smaller numbers of bits tending to be more common and ones with the answer bit set are relatively rare. The delta list is thus a Huffman code that can be looked up to find the answer bit and the number of bits in the delta which come immediately after it.

When validating a word the delta list is traversed without paying any heed to the answer bit. As another space saving measure, the Huffman tree is stored in a compact form requiring only 35 bytes and expanded at run time to Z-80 code to perform the Huffman decoding.

The answer bit can be used to extract all the answers but only in alphabetical order. A permutation is used to get them into required order. The permutation is conceptually a list of 2,313 numbers that say which answer number should appear in each position. These only need to be 12 bits in size since that's enough to refer to any of the 2,313 positions ($\log_2(2313)$ is 11.18). But these numbers can be reduced as we go along since each answer put in place means one less choice. The number of bits goes down to 11 when only 2047 possibilities remain and eventually drops to 0 bits as the last remaining possibility is the answer.

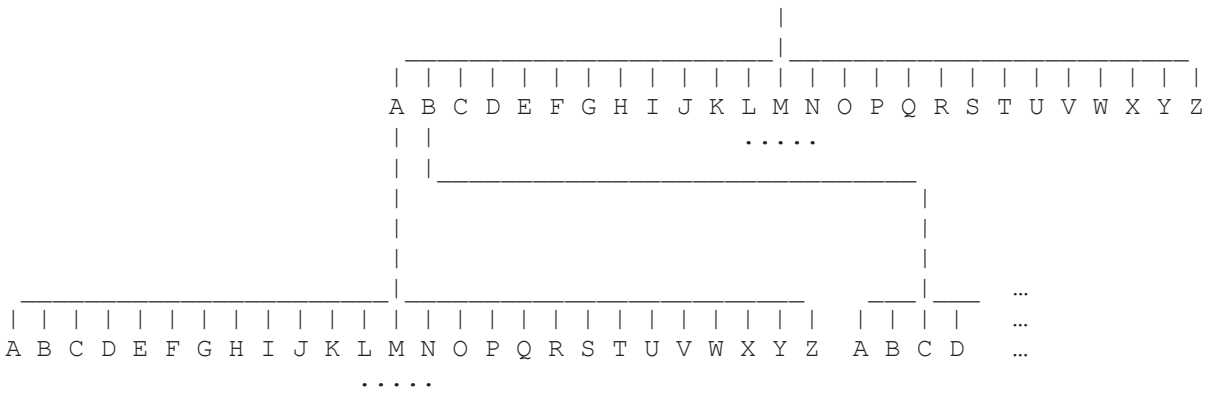
The code starts by unpacking the entire permutation list and then goes through the list swapping the current code with the one indicated in the permutation.

Once the list has been put into the correct order it only remains to look up what position answer N is in the alphabetical answer list and then scan the dictionary while counting down as we see each answer bit go by.

Alan Petrofsky

I'll be honest here in that I don't have a great understanding of this entry and this is more a summarization of the comments than independent thought. Nonetheless, it stores the entire dictionary in a list and the answer list is a compressed array of indices into the dictionary. As we've seen this normally requires 14 bits per index. Here the upper 6 bits are Huffman coded to reduce their size while the lower 8 bits are represented literally. I suspect this saves space as the upper bits can only be from 0 to 50 which is quite a bit smaller than the 64 values you could put in 6 bits.

The dictionary is represented as a walk over a 26-way, 5 deep tree of all words. The full tree is quite large. Here are two levels of it:



The tree is traversed depth first. You start at the top level with an empty word and ask if any words start with 'A'. Since there are you write down a 1 and, note that 'A' is the first letter and move down a level. Now you ask if any words that start with 'A' have 'A' as their second letter. Yes, so write another 1 and note we have two letters 'AA'. Are there any words that start with 'AAA'? No. That means the walk can stop going down and we write a 0. Since there are also no words that start with 'AAB' through 'AAG' we write 6 more zeros. At 'AAH' there is a word so we write a 1 and consider 'AAHA' which isn't valid so we write a 0 and will keep writing zeros until we hit 'AAHE'. We write a 1 for that and then 3 zeros because 'AAHEA', 'AAHEB' and 'AAHEC' are not words. Then we write a 1 because 'AAHED' is a word. And since we're at level 5 we know this is a valid word and there's no going down and simply proceed on to 'AAHEE'. None of the rest of the letters work so we write 22 zeros in a row. Having done all 26 letters we go back up and consider 'AAHF'. Once again no letters work and we do another 21 zeros in a row and go up a level and noticing that 'AAI', 'AAJ', 'AAK' are not the first 3 letters of any word so we have 3 more zeros written down. But 'AAL' does start a word so we write a 1 and consider 'AALA'. This continues on in a similar fashion (the next word will be 'AALII') until we get to the last word 'ZYMIC' and emit a bunch more zeros to “pop up” back to the top of the tree.

This is a fairly decent way to represent all the words as it only walks through parts of the tree necessary to reach all valid words. Many branches of the tree are ignored as they are nothing but dead ends.

Alan's code is a bit different as he starts at the end of a word and in goes in reverse alphabetical order. This is due to his compression scheme naturally producing the output in reverse order. I think the walk might be shorter working from the ends of the words but haven't checked that.

Since using Huffman coding is of no use on a two letter alphabet, a variant of ANS (Asymmetric Numeral System) coding called ABS (Uniform binary variant) is applied. Here's a reference for that:

https://en.wikipedia.org/wiki/Asymmetric_numeral_systems

Suffice to say that like Arithmetic coding it can approach the entropy of a symbol's probability when encoding. Or in other words it can use less than a bit to encode a symbol (the 1's and 0's of the tree walk, in this case).

To give better compression the probability of a 0 or 1 appearing is better predicted than just a simple average of the relative number of 0's and 1's. Instead quite a large number of tables are consulted based on the position in the tree and what letters precede the current one.

At the top level of the tree no prediction is needed since the dictionary has words that end in every letter (recall that we start at the end of the word and consider the 5th letter first). At the next level of the tree (the 4th letter) there are two predictions based on whether the 5th letter was a consonant or a vowel. The 3rd letter has 26 predictions based on the 4th letter. The 2nd letter has 2 * 26 predictions based on the 3rd letter and whether the 4th letter was a consonant or a vowel. Finally, the 1st letter has 2 * 26 * 26 predictions by having different ones for each of the 2nd and 3rd letter combinations and two for whether the 4th letter was a consonant or a vowel.

It's a pretty elaborate scheme and was a refinement of two earlier schemes which used fewer previous letters to make the prediction. Each prediction is the probability that the number is a 1 and is stored as a single byte covering 1/256 to 255/256. There are 2 + 26 + 2 * 26 + 2 * 26 * 26 = 1,432 prediction contexts requiring the same number of bytes of storage.

Unlike most of the entries, Alan chose to use zmac macros to produce the compressed data from the word list. Most often this work is done in an external program where it is considerably more convenient. I'll leave it to the curious to look for themselves to see how that rather unlikely and startling feat is accomplished.

All For Now

Phew, what a dazzling array of inventive and ingenious competition entries. I hope you found them as interesting as I did. While this will do for now there are still some developments to come. I hear Alan is working on a version of his program that will fit in 16K and run fast enough to be practical. I also secretly hope that someone out there will read this and plot some way to compress the dictionary even more. I'll be sure to let you know if there are any new developments. In the mean time, be sure to check out this year's competition!

TRS8BIT 2023 Competition

0K Z-80 Disassembler

A disassembler is a great addition to a debugger. It can show actual instructions instead of hexadecimal opcodes which, if you're like me, are quite tedious to disassemble by hand. Since a debugger must coexist with the program being debugged it should be as small as possible lest you get caught without enough space for the program and the debugger. Waste not, want not, as the old saying goes.

A Z-80 disassembler literally fitting in zero kilobytes is a tad unlikely. Nonetheless, there's every hope it can be squeezed into something less than 1024 bytes and will round down to zero kilobytes.

The smallest entry based on code size and run-time data space used will be the winner. You can use Z-80, 6809, 6803 or 68000 assembly to program your disassembler subroutine. All entries will be in source code form and will be published at the end of the competition for all to learn from and admire.

The winner will have their pick of any one of Ian Mavric's fabulous products for the TRS-80 line.

Competitors are encouraged to get in touch early in order to receive timely updates on leader board changes, rule clarification and supplementary materials. You are welcome to submit improvements to your entry multiple times.

For details on exact requirements, other rules and a running leader board please consult:

http://48k.ca/comp2023.html

All entries will be judged by George Phillips and
should be sent to george@48k.ca

Competition closing date is 15:00 hrs GMT,
23 November 2023.

TRS8BIT IS SUPPORTED BY YOU AND PRODUCED
AT WWW.TRS-80.ORG.UK

WHERE 8 BITS AND 16K STILL RULE OK