Apple-1 Keyboard Emulator based on the PC Printer Port

put together from various lab notes
by “Uncle Bernie” in November 2020

The need for a keyboard emulator

Every Apple-1 needs a keyboard, otherwise it's useless for anything, except maybe as a wall hanger. Various options for procuring a suitable keyboard do exist. Many Apple-1 builders cannibalize an old Apple-2, especially the early ones, which have become collectibles in their own right, so it's a pity to destroy them. You can also build the recently surfaced “open source” keyboard based on Cherry keyswitches:

https://github.com/osiweb/unified_retro_keyboard

Or use one of the various little adapter cards who plug into the Apple-1 keyboard socket and use standard USB or PS/2 keyboards.

All of these solutions work well but they may cost a lot of money. And not all offer an important feature which I call “auto-typing”. This allows you to automatically enter machine language programs into your Apple-1 with minimum effort. This feature is very useful during the bring-up phase of any new Apple-1 build. You don't want to type in memory test programs again and again by hand. “Auto-typing” relieves you from this burden.

This paper show you how to build an Apple-1 keyboard emulator yourself for minimum costs, using only off-the-shelf components. It comprises a keyboard adapter cable which first end plugs into the DB-25 line printer port of

an older PC or notebook computer, which you either have already or you can get one for a song. The other end of the cable plugs into the DIL-16 keyboard socket of your Apple-1. The PC / notebook should run DOS, and a very small and primitive program running under DOS receives keyboard entries and routes them to the Apple-1. Keyboard functions for the RESET and CLEAR SCREEN keys are provided. The auto-typing feature allows you to “download” any file into the Apple-1 with minimum effort. The format for this file is the usual WOZMON command syntax, the same method as used in the POM1 Apple-1 emulator, so all the software and tools you wrote for POM1 can be used on the real deal.

Note that this is what I consider a “cheap hack”. It is not a viable commercial product that could be sold: it takes about one hour to put one together and it only runs on obsolete hardware. Even at minimum wage, the result would be too expensive to be sold with any reasonable profit. The time to make one could be reduced to about 10 minutes by using a small PCB and press-fit connectors throughout, but the worldwide Apple-1 crowd is to small to develop that.

So accept that this is a hobbyist's (me) cheap hack for hobbyists. If you want to make more to sell them, you are welcome, I don't care, all the tips, tricks and software listings I post on Applefritter are public domain, and use them at your own risk.

Actually, how this keyboard cable came about was that after finishing my first Apple-1 build, I had no ACI yet, and as the damn thing always crashed on me, as I did not have found all the reliability mods, did not want to enter the DRAM test program by hand again and again and again. When I had the auto-type part going, I stopped to further develop this software. So it has nasty fixed timing loops instead of using the clock functions from the C library, etc., and you probably need to hack the source code a bit to make it work on faster computers. Of course I could write a version that does everything automatically, finds the port, and can be configured to compile on every known operating system on every known machine. Alas, I don't have the time for that.

Required components

1 x DB-25 connector, male, preferably press fit type for flat band cables, but the solder bucket type also is OK.

1 x length of 25 pin flat band cable (or any 15 wire cable, or 14 wires plus shield, if no press fit connector is used) You can use any reasonable length, but I tried only up to 3' of cable length.

1 x DIL-16 header with small diameter pins on one side and solder forks at the other side.

1 x small NPN transistor of any type.

~5 inches of heat shrink tubing, of a small enough diameter that it just slips over the solder side pins of the header without effort.

A small piece of heat shrink tubing that loosely slips over the transistor body.

One ancient PC or notebook with LPT port running DOS.

How to build the keyboard emulator cable

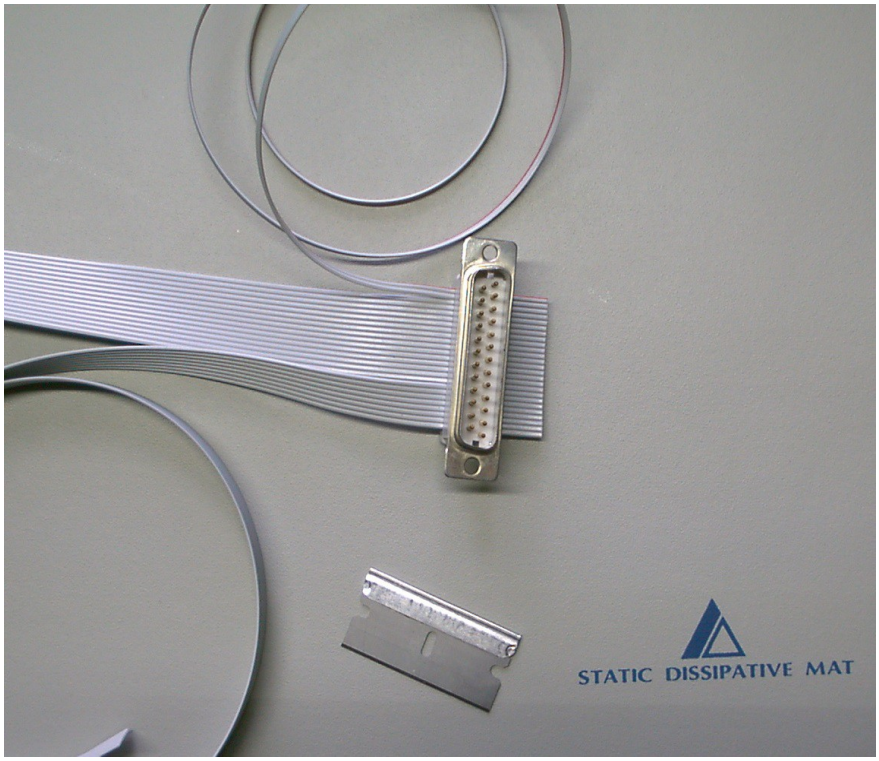
Steps 1 to 5 are for press-fit DB-25. For any other cable / connector combination, just do the usual wire stripping and soldering into the right pins of the DB-25, as shown in the schematic below. If you use a shielded cable with 14 wires plus shield, use the shield as ground return (DB-25 pin 18) and not as a signal line.

Step 1: Using a vise, or the correct press-fit-tool, press on the flat band cable to the DB-25 connector. Make sure the cable is aligned properly before tightening the vise. Do not over-tighten as this may crush the connector shell.

Step 2: Separate the first 2 and the last 8 wires of the flat band cable. This is best done using a scraper blade: cut into the flat band cable on the right places to start, then carefully and slowly pull the three groups of wires

apart, and whenever the insulation ripping apart would to start to run away from the thin membrane connecting the wires, and begins to run into the thicker insulation around the wire, use the scraper blade again to help it go back to the thinnest membrane. Some brands of flat band cable separate fine without such problems, but others don't.

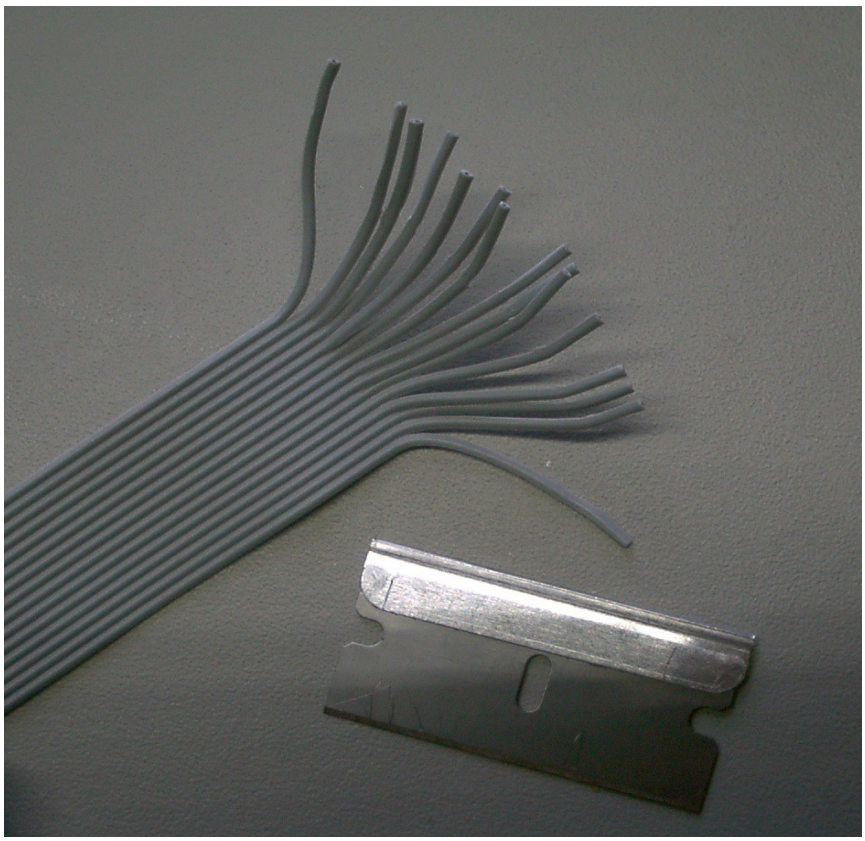
This is the result after the three groups have been separated:



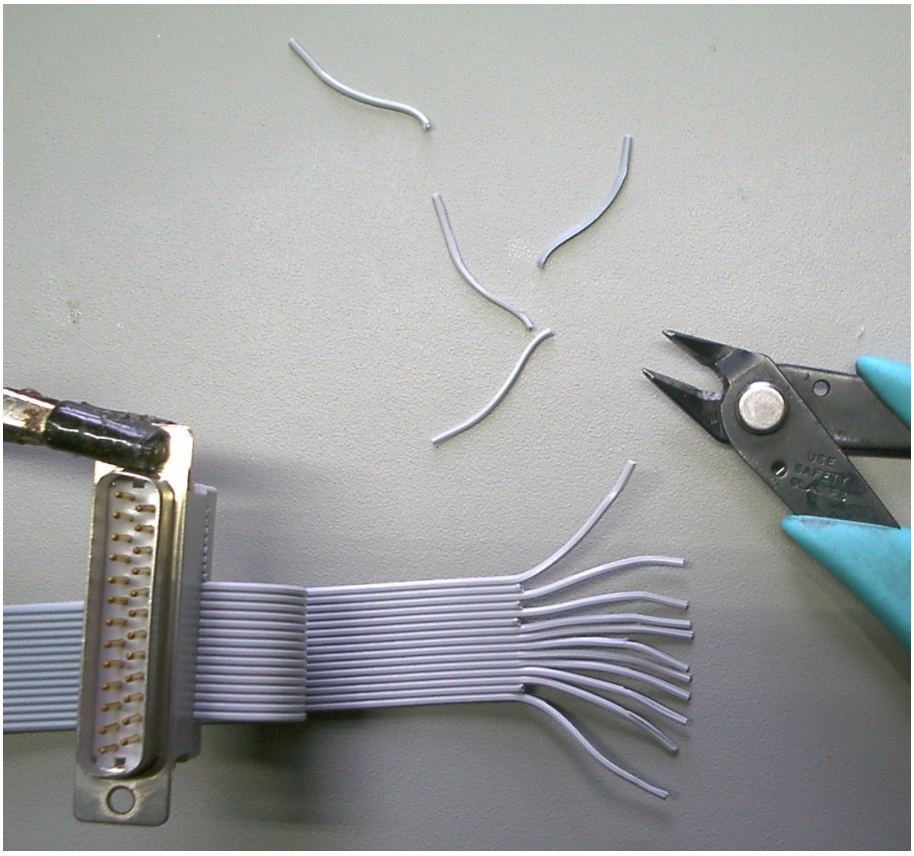
Step 3: Cut away the portion of the flat band cable that sticks out of the DB-25 connector. Also cut away the separated wire groups of 2 and 8 wires, keeping the middle group. Be careful not to cut into the remaining wires of the middle group or into your fingers.
A knife edge held perpendicular to the connector at the right place can protect the wanted wires of the middle group from the cutting operation, which works best if the DB-25 is held in a vise.

This is the result of step 3:

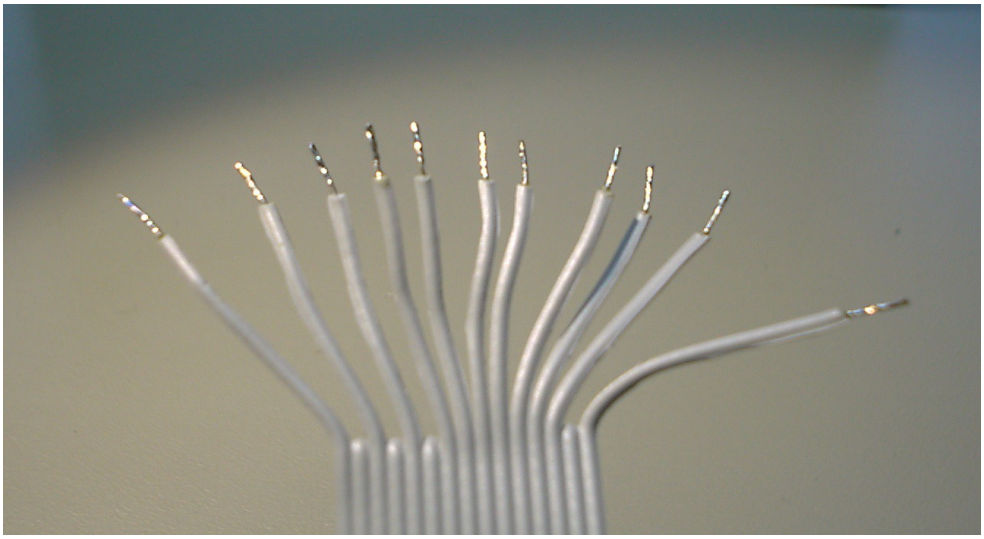
Step 4: Again using the scraper blade, separate the wires on the other end of the flat band cables to a length of 1 inch. Some types are nasty and their wires can't be pulled apart without severing the insulation, which is bad. Those need to be cut wire by wire.



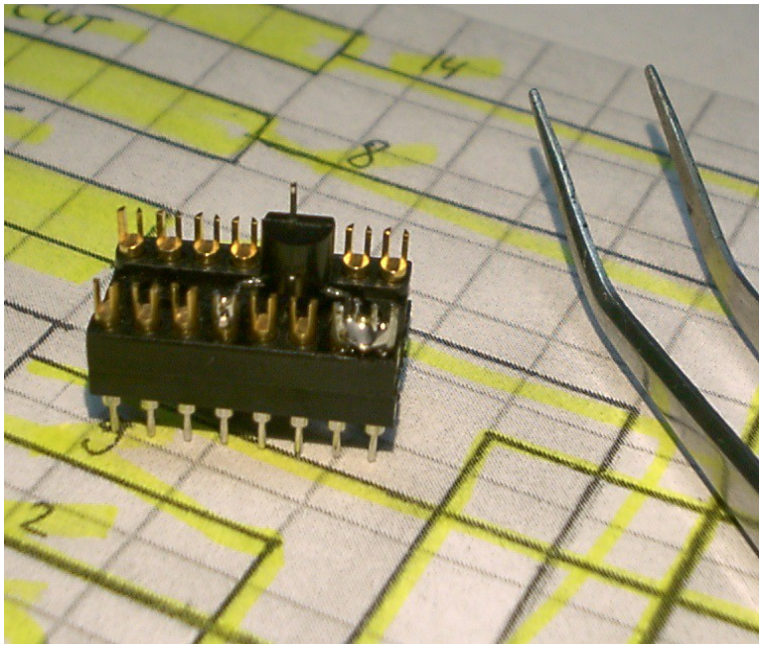
Step 5: Snip away the 2nd, 10th, 12th, 14th wires. The count starts at the side where the 2 wires were removed in step 3. You can consult the schematic below to be sure before snipping the wrong ones away.

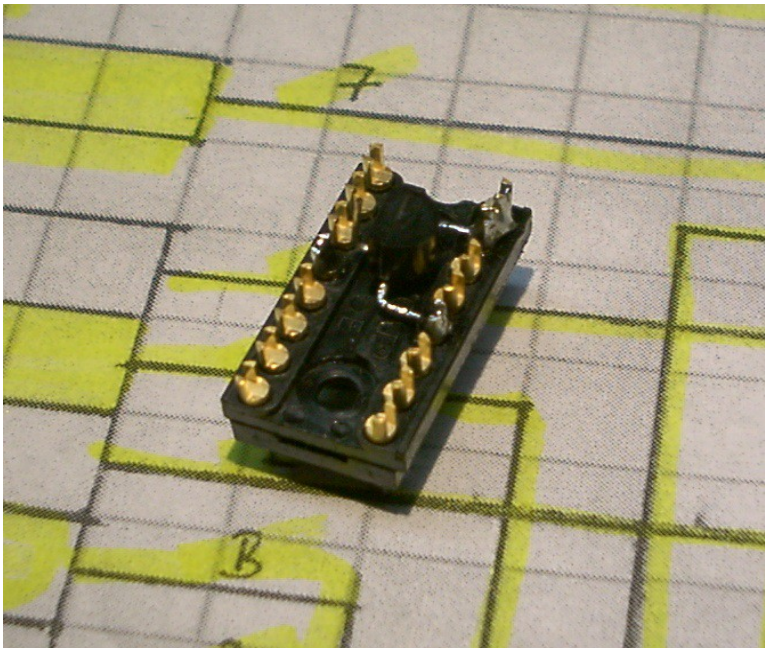


Step 6: Strip the insulation on the wires by about 1/8th of an inch, twist the strands, and solder them to be rigid.



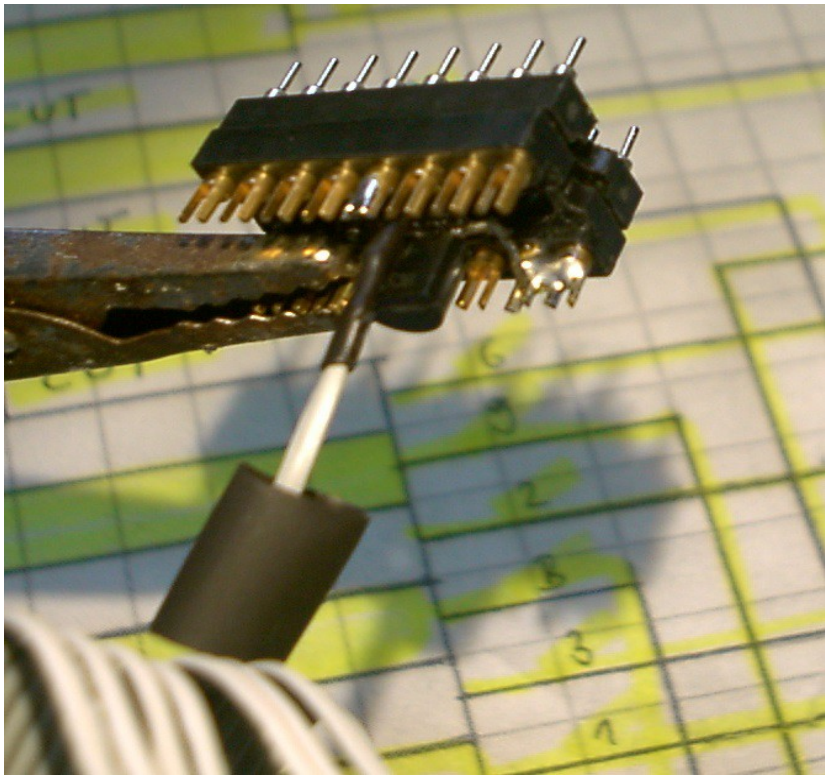
Step 7: Solder the transistor into the header as shown in the following photos. Make sure the EBC sequence is correct as per the schematic below. It is advisable to protect the thin pins of the header during all the work by inserting them into a DIL-16 socket, as seen here in the photos:



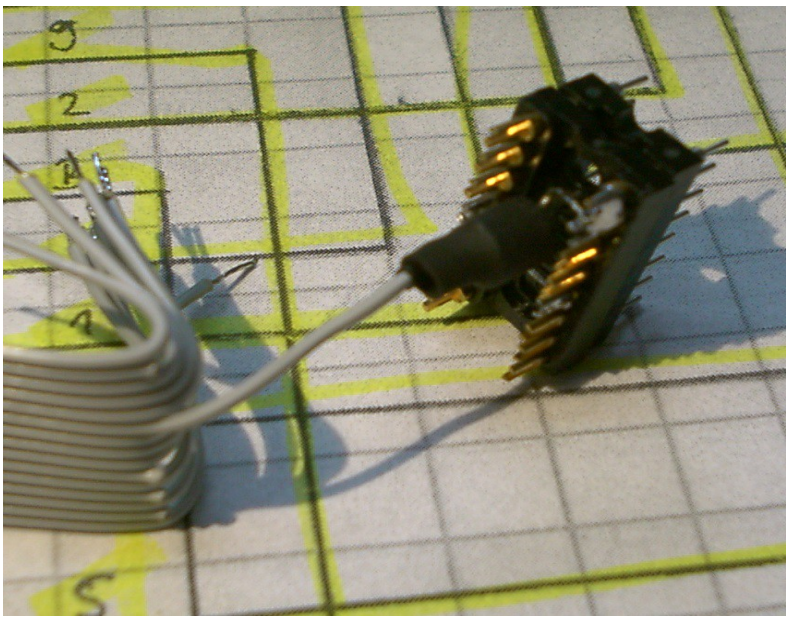


Step 8: Slip one small diameter heat shrink tube piece and the one larger diameter heat shrink tube piece over the 6th wire that goes to the transistor base (DB-25 pin 17).

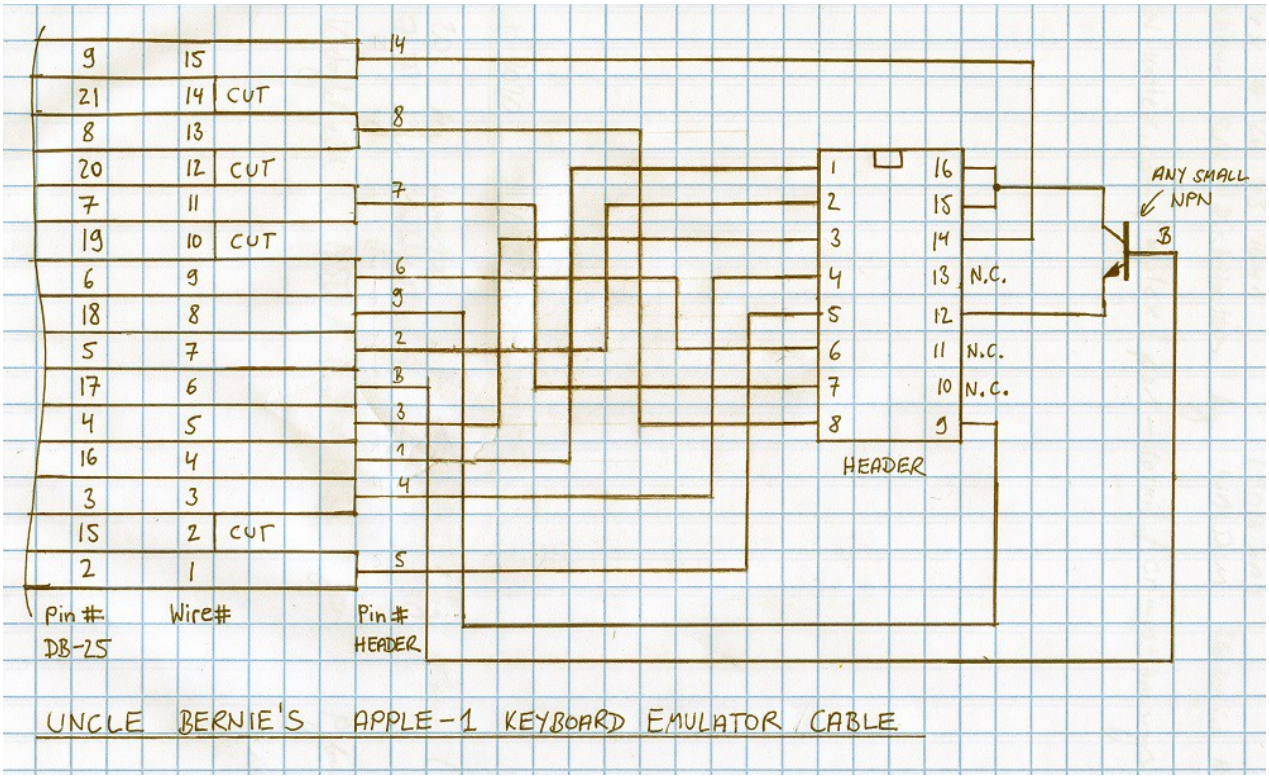
Solder the wire to the transistor base. Move the small diameter heat shrink tubing over the solder joint and shrink it in place using a heat gun:



Step 9: Slip the larger diameter heat shrink tubing piece over the transistor body and shrink it in place.



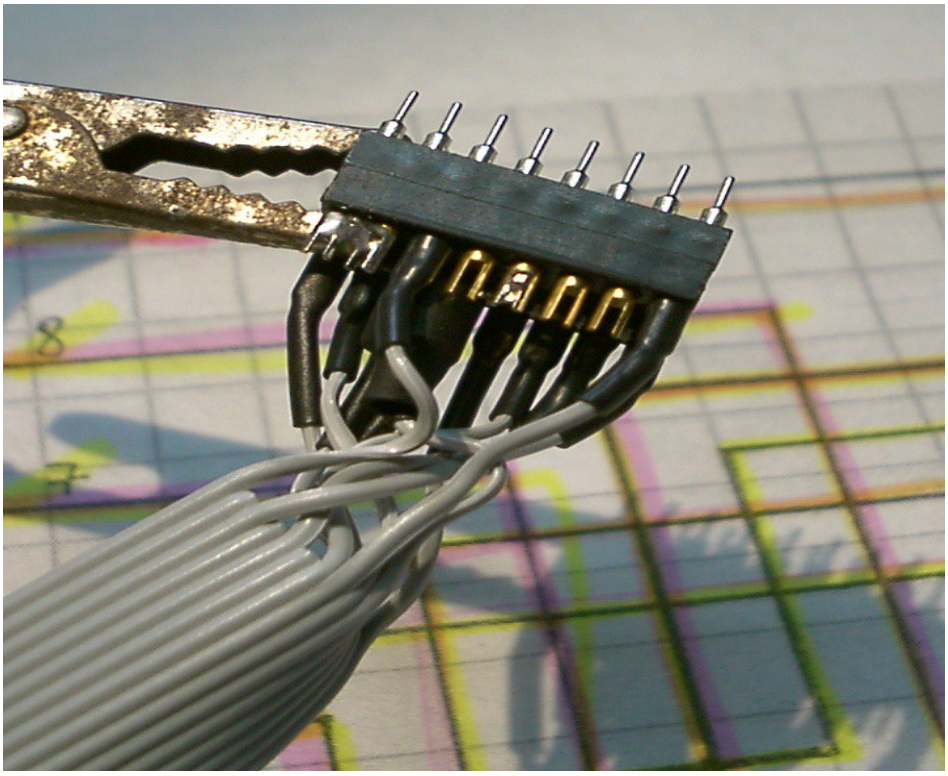
Step 10: Solder the wires to the header one by one according to the following schematic. Do not forget to slip a short (3/8”) piece of heat shrink tubing on each wire and push it up the wire as far as possible away from the solder joint before soldering. We do not want the shrink process to start yet, so keep them away from the soldering heat.



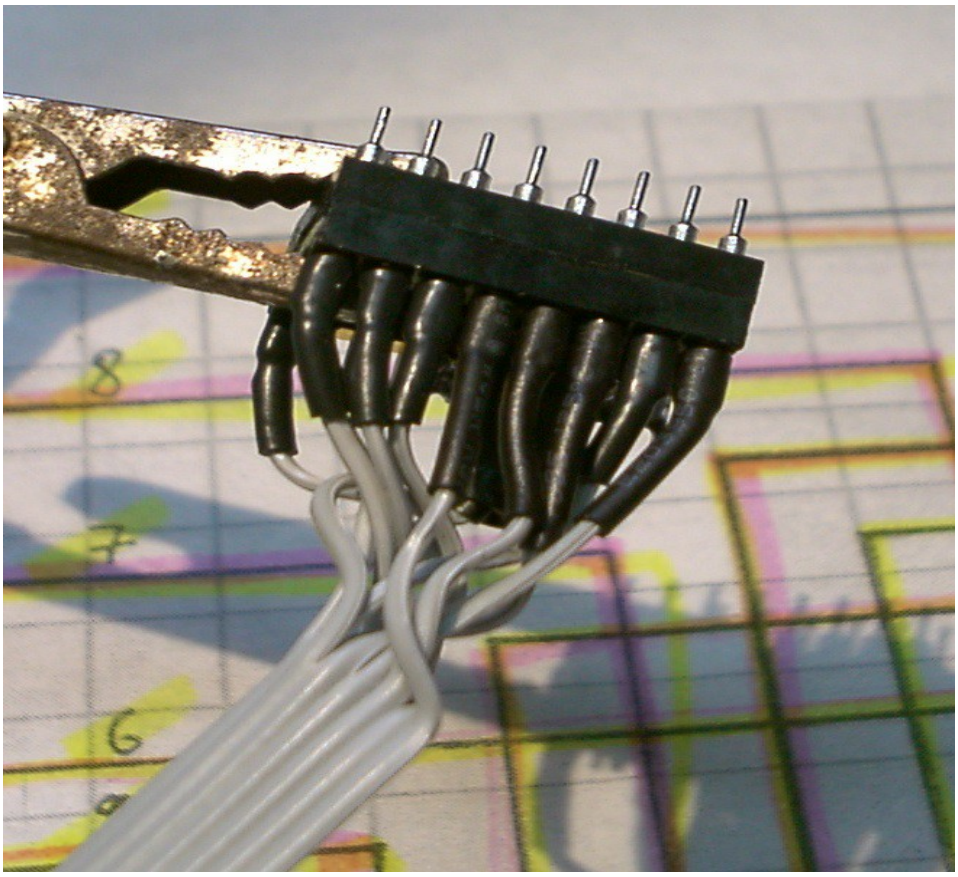
Step 11: Put the connector in a vise, pins up, and secure the header in some other clamp. Use a multimeter to verify all connections are correct according to the above schematic. Correct any mistakes before the next step.

Step 12: Slide all the heat shrink tubes down towards the header and over the header solder joints. This requires some patience and bending some adjacent wires out of the way where they block the slide operation. Then use a heat gun to shrink the heat shrink tubes into place. They are essential for a long life of the cable.

The result from step 12 should look like this:



and like this (from the other side):



The keyboard emulator software

This chapter shows the keyboard emulator software source code listing. It runs under DOS, and it should compile with any ancient C compiler. I used Microsoft C7.00 which I have bought back in the 1980s. Unlike other Microsoft tools, which IMHO are mostly junk, it never failed me. This is a great compiler.

It should be possible to add a few lines of code and modify a few library calls to make it compile and run under Linux, but I'm too busy to give it a try. The drawback of using Linux for this kind of hardware level work is twofold:

first, to gain direct access to the hardware, such programs must run with root permissions, which is easy to do as it is dangerous:

```
chown root:root <filename>
chmod a+s <filename>
```

and second, Linux is a preemptive multitasking operating system so you can't do timed bit-banging on the ports in any easy way, the only way I know is to put such timing critical code into the kernel.

Timed bit-banging is not needed for this keyboard emulator, but for other things I like to do timing loops. So I never did get around to use Linux for all this direct port work, and always used DOS for this, which is really nice for this because it is primitive enough. In DOS, you can even turn off or re-route all the hardware interrupts as you please with only a few lines of assembly code ! Any such code can take complete control of all the hardware in the computer at any time and even kick DOS out of the machine entirely.

This is not only a pathetic vulnerability, but also a great feature of these ancient computers and their long obsolete “toy” operating systems which has been lost in time, due to the constant war against viruses and worms and other nasty malware. But you would never plug your ancient, 100% virus-free notebook running DOS that lives in your electronics

lab into the internet, won't you ?

See my point: even these old, long obsolete systems that can be had for a song can still be useful in the 21st century. They offer a much easier and less frustrating path towards makeshift hardware-level bit banging as even the nicest Raspberry Pi could ever do. And they cost even less!

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <conio.h>

#undef DEBUG              /* turn on to get more verbose and detailed output */

/* Width of STROBE pulse. On the Compaq Aero 4/33C, 0.1 sec for 7700 ticks */
#define DELAY 7700

#define BACKSP 0x5f
#define POS1   0x02

typedef unsigned char byte;
typedef unsigned int  word;

word lpt_ad = 0x378;

#define RD_DPORT()  _inp(lpt_ad)
#define WR_DPORT(x) _outp(lpt_ad, x)
#define WR_CPORT(x) _outp(lpt_ad + 2, x)

void do_cport(int idle, int active)
{
 unsigned long i;

 WR_CPORT(idle);
 for(i = 0; i < DELAY * 81; i++) WR_CPORT(active);
 WR_CPORT(idle);
}

#define FILENAMLEN 16

int main(int argc, char *argv[])
{
 int i, j, key, zflg;
 FILE *fp;
 char fname[FILENAMLEN + 1];

 WR_CPORT(0x0c);
 zflg = 0;
 fp = (FILE *) 0;

 while(1) {
  if(fp) {    /* if file is open, get characters from file, not from keyboard */
   key = fgetc(fp);
   if((key == EOF) || _kbhit()) {
    fclose(fp);
    fp = (FILE *) 0;
```

```
    continue;
   }
   /* process special characters here, may end with continue */
   if(key == 0x0a) key = 0x0d;                          /* convert LF to CR */
  } else key = _getch();        /* no file open, get characters from keyboard */
  if(key) {                             /* if key code != 0, emulate key press */
   if(zflg) {               /* this <key> code had a preceeding zero key code */
    zflg = 0;
    if(key == 0x3b) {
#ifdef DEBUG
     printf("\nF1: RESET\n");
#endif
     do_cport(0x0c, 0x08);
     continue;
    } else if(key == 0x3c) {
#ifdef DEBUG
     printf("\nF2: CLR SCREEN\n");
#endif
     do_cport(0x0c, 0x04);
     continue;
    } else if(key == 0x3d) {
     if(fp) fclose(fp);
     do {
      printf("\nEnter filename: ");
      if(!fgets(fname, FILENAMLEN, stdin)) continue;
      j = strlen(fname);
      if(fname[j - 1] == '\n') fname[--j] = '\0';
     } while(!j);
     fp = fopen(fname, "r");
     if(fp == NULL) printf("Sorry, can't open file '%s'", fname);
     putchar('\n');
     continue;
    } else if(key == 0x53) key = BACKSP;
    else if(key == 0x4b) key = POS1;
   }                         /* end of key codes with preceeding zero key code */
#ifdef DEBUG
   printf(" %.2x", key);
#endif
   if(islower(key)) key = toupper(key);
   key &= 0x7f;
#ifndef DEBUG
   putchar(key);
   if(key == 0x0d) putchar('\n');
#endif
   WR_DPORT(key);
   key |= 0x80;
   for(i = 0; i < DELAY; i++) WR_DPORT(key);
   key &= 0x7f;
   for(j = 0; j < 10; j++) {
    for(i = 0; i < DELAY; i++) WR_DPORT(key);
    if(fp == NULL) break;
   }
  } else zflg = 1;                      /* if key code == 0, memorize this. */
 }

 exit(0);
}
```

Note there are a few potential issues you may want to fix before this runs on your machine:

All the timing hinges on the DELAY constant, which then is used in the do_cport() function which repeats the bit-banging function call every so often. This is weird programming at the first glance but it helps to control the timing being more predictable. Depending on the particular machine, if you do the write to the port only once, and then do some sort of timing loop, this timing loop a) might be removed by the optimizer stage of the C compiler, and b) might execute in a cache memory and then be much faster than you want it to be. Accessing the port again and again and again slows the whole thing down in a much more predictable manner, even across a multitude of machines of the same era. Quick and dirty, but it works. I wrote and debugged the whole code in less then one hour, starting from scratch.

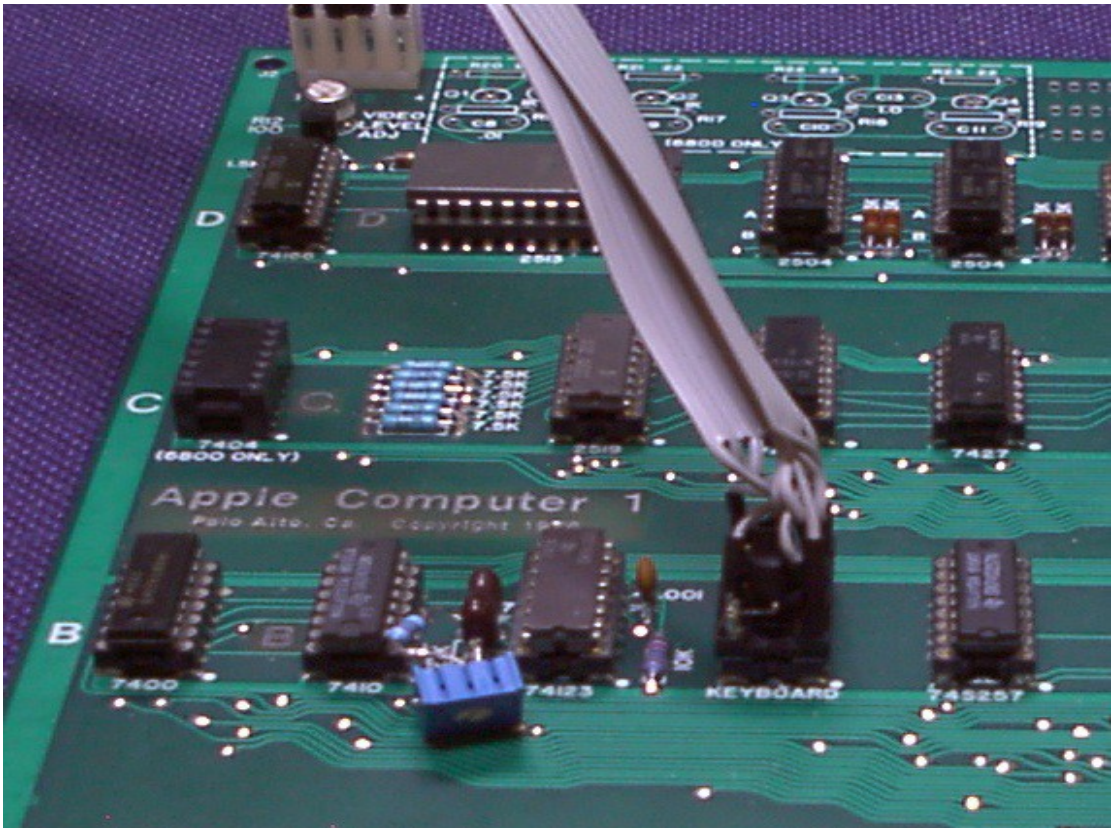
The port address is assumed to be 0x378, which is true for many ancient machines, but may not apply to yours. Change the lpt_ad if need be.

All I/O operations should have been routed through the macros RD_DPORT(), WR_DPORT() and WR_CPORT() which prepares a possible future port to Linux. But I'm not sure this was done completely and would work.

Data types byte (8 bit) and word (16 bit) are tailored to get properly sized address and data parameters for the I/O library functions. These typedefs must be adapted for any machine architecture that has different sizes of the elementary data types. More modern compilers offer header files that provide data types by size, with standardized names, but the ancient machines I use don't offer that.

Use of the keyboard emulator cable

With the Apple-1 power supply switched off, plug the DB-25 into the LPT port of your computer running DOS. Plug the header on the other side of the cable into the Apple-1 keyboard connector DIL-16 socket at location B4:



(Note that this Apple-1 build still has the /CAS timing trimmer mod which allows finding the best setting for the /CAS timing)

Check and double check that the header was inserted correctly. If it is inserted backwards or with an offset leaving some pins unconnected, it may cause damage to your computer or the Apple-1.

Then start the a1kb.exe program and power up the Apple-1. Initially you probably want to clear the screen and then give a RESET to start WOZMON:

Key F1: RESET
Key F2: CLEAR SCREEN

Then you can type around to see if all characters on the keyboard end up on the screen. This is a nice feature of the WOZMON, but be aware if you spill over its input buffer, there will be an extra backslash character. The ESC key should have the same effect. This is what I typed in:

After these tests are done and your keyboard works, your Apple-1 is good to go as it now has a “keyboard”. And a very powerful one, too:

With the F3 key, you can open any file on the computer running the keyboard emulator, which then will be transferred character by character to the Apple-1 using the WOZMON, so the file must use the WOZMON command syntax. If this auto-typing runs too fast, WOZMON can't catch up and will drop characters, leading to erroneous results. The bottleneck is the one character per frame limit of the Apple-1 terminal section: WOZMON will always wait for the character being accepted, and then looks for a keyboard entry, and so you can't run auto-typing too fast.

If the auto-typing runs too fast, change the DELAY constant the keyboard emulator source code and recompile.

Good luck and have fun !