Bildspeicher mounten

Aus meiner Zeit bei Robotron. — Rolf Böhm, 08.04.2020

Eine Comoputertomografie von einer lebenden Festplatte

Das Schöne am Programmieren ist, dass man mit Computern immer wieder Wunderdinge vollbringen kann, Sachen, die eigentlich völlig unvorstellbar sind. Eines der eindrucksvollsten Wunder, die ich in 40 Jahren Programmierpraxis erlebt habe, ist das Bildspeicher mounten dank eines Geniestreiches von Andreas Graf vom Zentralinstitut für Kybernetik und Informationsprozesse (ZKI) an der AdW der DDR. Auch wenn es lange her ist und es diese Rechner überhaupt nicht mehr gibt – das muss ich einfach einmal loswerden.

Das Bildverarbeitungssystem Robotron A6472 mit Steuerrechner K1630 unter dem Betriebssystem MOOS 1600, war eine Entwicklung des ZKI und Andreas Graf hat den Bildspeicherdriver geschrieben.

Externe Geräte wurden über Driver angesprochen. Die Geräte wurden mit zwei Buchstaben gekennzeichnet, dann gab es noch eine Nummer, um einzelne Laufwerke zu unterscheiden und am Ende einen Doppelpunkt als „Gerätekennzeichen[1]“. MT0: etwa war das Magnetbandlaufwerk 0 (Magnetic Tape), LP1: der Drucker 1 (Line Printer) und DK0: (wie Disk 0) war die System-Festplatte. Die zwei Buchstaben waren zugleich der Drivername. Die Magnetbandstationen hat der MT-Driver angesteuert, der Drucker hatte einen LP-Driver und die Festplatte wurden per DK-Driver angesprochen.

Was der Driver zu machen hatte, bekam er vom Anwenderprogramm über sog. QIO-Funktionscodes mitgeteilt. IO.RLB hieß „read logical block“, IO.WLB „write logical block“. Ein Datenblock war meist 512 Byte lang.

Bei Geräten, die Dateien speichern konnten, gab aber eine Besonderheit. Die logischen Blöcke waren fortlaufend pro Gerät gezählte Blöcke (auch als „physische Blöcke“ bezeichnet[2]). So eine Festplatte hatte also vielleicht 4.000 logische Blöcke, Dateien spielten hierbei keine Rolle. Für Dateien gab es eine zweite Nummerierung, „virtuelle Blöcke“, die wurden je dateiweise ab Dateianfang gezählt. Und für die gab es zwei weitere QIO-Funktionscodes, IO.RVB „read virtual block“ und IO.WVB „write virtual block“.

Das Dateisystem musste vorbereitet werden. Zunächst musste die Festplatte *initialisiert*[3] werden, d. h. auf dem Datenträger war – einmalig – ein leeres Dateisystem zu erzeugen. Das erfolgte mit dem MCR-Kommando (Konsolenkommando) INI. Aufpassen, INI war brandgefährlich, anschließend waren alle alten Dateien futsch. Beim Initialisieren wurde u. a. das berühmte Indexfile erzeugt, die INDEXF.SYS. Wenn die Platte dann initialisiert war, musste sie in der jeweiligen Sitzung *gemountet* werden. Nun konntest du nach Herzenslust

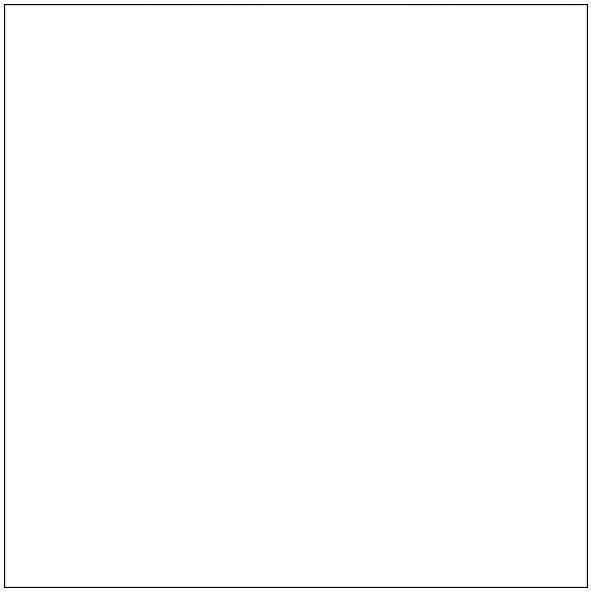
[1] Ist ja auch heute noch beim PC so, siehe „LPT1:“.
[2] Einziger Unterschied, die physischen Blöcke begannen man bei Null zu zählen, die logischen Blöcke bei Eins.
[3] Formatieren lassen wir mal außen vor.

Dateien anlegen, lesen, schreiben, löschen usw. — kurz nach Herzenslust virtuell Lesen und Schreiben[4].

Nun war unser Rechnersystem ja ein *Bildverarbeitungssystem,* d. h. es gab Bildspeicher und zwar 8 Stück. In denen standen die Bilder für den Displayprozessor. Für den Bildspeicher, gab es natürlich auch einen Driver. Das war der RM-Driver RMDRV.SYS, RM wie „refresh memory“, Bildwiederholspeicher. Die 8 Bilder waren im „Systemjargon“ dann die „Geräte“ RM0: … RM7:.

So ein Bild hat natürlich mit Dateien eher nichts am Hut und folglich war da ein virtuelles Lesen oder Schreiben eher kaum sinnvoll. Als Andreas Graf die Aufgabe erhielt, den Driver zu schreiben[5], hatte er aber dennoch die Chuzpe die Funktionscodes IO.RVB und IO.WVB zu implementieren. Das war genial.

Nach dem Einschalten des Rechners waren Bilder gewöhnlich leer und sahen so aus:


Leeres Bild.

Doch nun gab es eben das virtuelle Schreiben. So ein System ist weiß ja nicht wirklich, was Festplatten und Bildspeicher sind. Es kennt nur die je zwei Driverbuchstaben, einmal eben DK, einmal eben RM und auf denen waren eben Datenblöcke zu je 512 Byte.

Die Festplatte wird mit dem Kommando „INI DK0:DATA“[6] initialisiert. Wir wissen nicht genau, wie da im Einzelnen das Indexfile (und ein paar andere fundamentale Dateien) angelegt werden, sicher ist aber, es läuft über den DK-Driver und er liest und schreibt da virtuelle Blöcke[7].

---

[4] Logisches Lesen und Schreiben war übrigens ab dem Mounten verboten. Das leuchtete ein, denn wenn man da infolge eines Programmierfehlers versehentlich die INDEXF.SYS überschrieben hätte, wäre ja das ganze Dateisystem futsch gewesen.

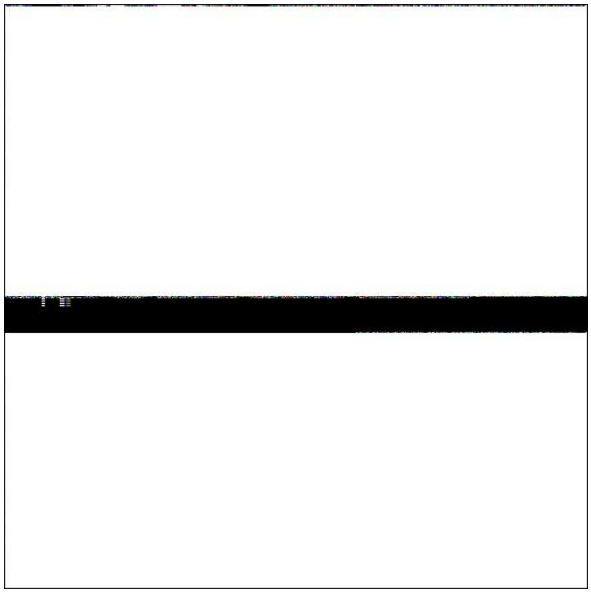
[5] Bei uns bei Robotron hat Heinz Pahlke den RM-Driver beschraubt und mir das alles erzählt.

[6] „DATA“ ist ein Pflichtparameter – das ist der Name, den der Datenträger erhält.

[7] Außer dem Driver spielen da auch noch Hilfssteuerroutinen, die ACP eine wichtige Rolle, das ist aber ein anders Thema.

Was würde wohl passieren, wenn wir, sagen wir mal, auf einem Drucker versuchten, ein Dateisystem zu initialisieren, etwa mit „INI LP1:DATA“? Na, da käme aber eine fette Fehlermeldung, denn wie soll das gehen, Daten von einem Drucker einzulesen.

Wie wäre es aber einmal mit einem „INI RM0:DATA“? Jetzt geschieht ein Wunder. Es flimmert ein bisschen an dem Farbmonitor, der die Bilder anzeigt, dann sieht das Bild so aus:


In der Bildmitte: Das Indexfile INDEXF.SYS.

Der dunkle Balken in der Bildmitte ist nichts Geringeres, als das Indexfile, das das Dateisystem konstituiert. Und da sind auch schon einige Bytes drin. Blockadresszeiger vom Master File Directory etwa. Der Bildspeicher speichert 512 Zeilen à 512 Byte und jede Zeile ist für den RM-Driver ein „Festplattenblock“. Dass Indexfile in die Mitte des Bildes hat übrigens seinen Grund: Das kommt bei den Festplatten der Lesekopf am schnellsten hin. Wer genau hinsieht, bemerkt auch, dass die alleroberste Zeile ein wenig flimmrig ist. Das ist der Bootblock[8], den das INI ebenfalls erzeugt.

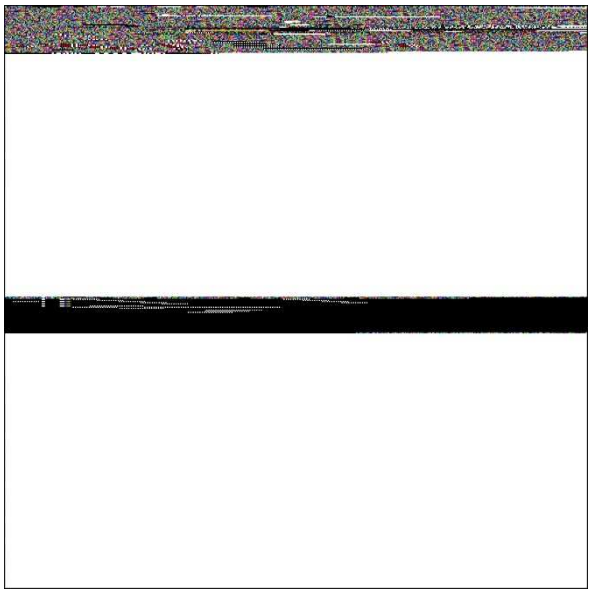
Der nächste Schritt ist nun, Bildspeicher zu mounten: Das macht das MCR-Kommando „MOU RM0:/OVR[9]“. Wird es funktionieren? Es funktioniert: Erneut ein kurzes Flimmern – erledigt. Der Bildspeicher ist jetzt eine „Festplatte“, auf der man „ganz normal“ Dateien speichern kann.

Du kannst nun irgendein beliebiges Programm laufen lassen und damit auf „der Festplatte RM0:“ Dateien anlegen. Der Computer merkt ja überhaupt nicht, dass Andreas Graf ihm da „unter dem Hintern weg“ eine Festplatte durch einen Bildspeicher ersetzt hat.

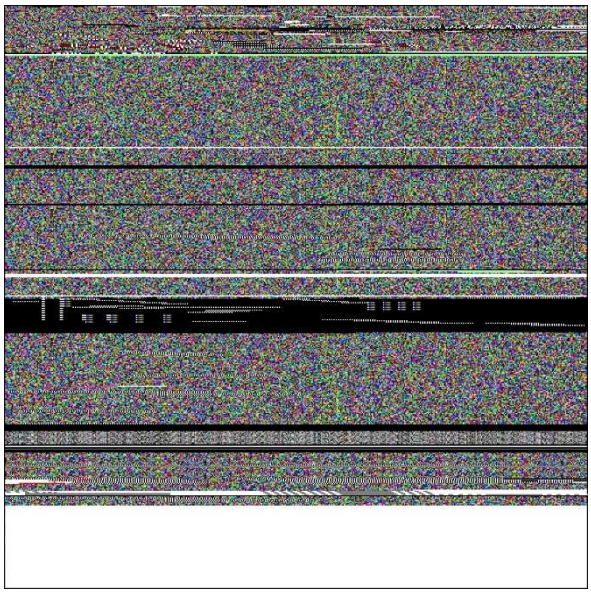
Das eigentliche Wunder ist aber, dass der Inhalt des Bildspeichers ja zugleich als Monitorbild angezeigt wird. Plötzlich kannst du da *an deinem Monitor* sehen, wie er da rumschwubelt. Die Dateibytes sind ja zugleich Pixel. Und wenn er da nun plötzlich Dateien in das Bild reinschreibt, siehst du, wie sich das Bild sich allmählich von oben nach unten mit Pixelstrukturen füllt:

---

[8] Genau genommen ist es kein richtiger Bootblock, sondern lediglich ein „Pseudobootblock“, der bei Bootversuch „THIS VOLUME DOES NOT CONTAIN A HARDWARE BOOTABLE SYSTEM“ auf Monitor ausgibt.
[9] Mounten am besten immer mit dem Schalter /OVR.


Er fängt oben an, Blöcke zu schreiben.

Nach und nach schreibt dein Programm die „Festplatte“ immer mehr voll:



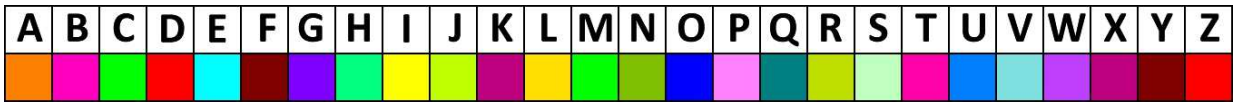
Wenn die unterste Bildzeile erreicht ist, kommt folgerichtig auch die Systemmeldung Festplatte RM0: voll. Da gibt es dann nur eins: Du musst ein paar Dateien löschen.

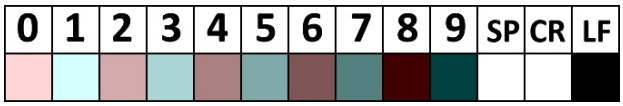
Dabei entstehen Lücken, in die er dann neue Dateien reinschreit, die „Festplatte“ fragmentiert sich allmählich. Was du das siehst ist, ist eine Art *Computertomographie von einer „lebenden Festplatte“*.

Das war aber nicht nur eine Spielerei. So hatten wir immer „ein paar Arbeitsplatten“ außer der Reihe zur hand. Okay, auf die Dinger passten nur knapp 1/4 Megabyte, dafür waren sie blitzschnell. Weder rotierte da eine Platte, noch gab es irgendwelche Schreibköpfe zu positionieren.

Assemblerläufe dauerten manchmal ewig, wenn du zu viele Makros benutzt hast. Einfacher Trick: Die Makrobibliotheken vor dem Assemblieren in den Bildspeicher reinkopieren, und schon ging es 10× schneller.

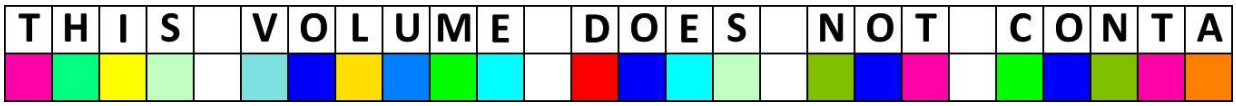
Der besondere Clou war eine Lookuptabelle mit den ASCII-Codes der Buchstaben in schön unterscheidbaren Farben. Die Vokale blau und gelb, die Konsonanten in rot, grün, braun, etwa so:





Kryptische Codes codieren und entziffern, das bringt man ja als Programmierer. Nun das Bild am Monitor bissl vergrößern und schon konntest du direkt lesen, was auf deiner „Festplatte“ stand. Mit etwas Üben hatte man das nach einer Weile ganz gut raus. Ich kam mir vor wie Turing beim Entschlüsseln der Enigma.

Zum Beispiel vorn im Pseudobootblock:



THIS VOLUME DOES NOT CONTAIN A HARDWARE BOOTABLE SYSTEM.

— — —

Einen Nachteil hat es aber dennoch gehabt. Die Bildspeicher waren ja Halbleiterspeicher und keine Magnetspeicher. Strom ausschalten und alles ist futsch. Also vorher immer schön – Daten sichern.

Heute heißen solche Dinger übrigens Solid-State-Drive oder USB-Stick. Alles nichts Neues. Diese Technik gab es dank Andreas Graf vom ZKI bei Robotron schon 1984.

— — — — —