Die RSX-11 ohne FCS-Geschichte

Software-Geschichten aus alter Zeit. Aus meiner Zeit bei Robotron. — Rolf Böhm, 16.04.2021, 11.10.2022

Bei Robotron

Wir hatten damals bei Robotron den K1630 zu programmieren. Das Betriebssystem war MOOS 1600, aber es war ein offenes Geheimnis, dass das in großen Teilen das RSX11 von der DEC PDP11 gewesen war. Programmiert wurde selbstverständlich in Assembler. Ganz fundamental in jedem Betriebssystem ist das Dateisystem. Dateien lesen und schreiben. Dateien müssen aber auch angelegt, gesucht, geöffnet, umgenannt, geschlossen und gelöscht werden.

Dafür hat das originale RSX die äußerst komplizierten, ungeheurer viel Speicherplatz und Laufzeit fressenden *File Control Services* FCS bereitgestellt. Das waren gewaltig kryptische, Makros und Unterprogramme mit Unmassen Fehlermöglichkeiten.

Eine ganze Generation von Programmierern hat von diesem FCS nur mit allerhöchstem Respekt gesprochen. Und ohne FCS bist du als Anwendungsprogrammierer an keine Datei rangekommen. Schlimm, schlimm.

Wir bei Robotron hatten immer eine ungeheure Hochachtung vor den Systementwicklern der DEC, Maynard, Massachusetts. Doch auch die haben manchmal gepfuscht. Das FCS war, behaupte ich einmal, einfach schlecht gemacht. Schnell zusammengeschrieben, anschließend der Pfusch mit den Makros bissl überdeckt. Indem die Programme das FCS genutzt haben, sind sie unnötig groß und langsam geworden und auch das Assemblieren hat gedauert.

Der Systemkernel war solide. Dateien, das musste auch irgendwie anders gehen, ohne FCS. Aber wie da rankommen? Dokumentiert war da nichts. Um das rauszukriegen muss man ganz tief in das System eintauchen. Und dann gab es ja noch diese geheimnisvollen ACP.

Es musste auch irgendwie anders gehen

Das FCS waren auch nur so Unterprogramme, kein irgendwie höherer System-Hokuspokus. Das war auch nur so gelinkte Code in deiner Exe. Oder besser, deiner Task, Exen hießen in dem System Tasks. Was steckte da dahinter?

Dazu ein wenig Vorwissen.

1. Block Lesen und Block Schreiben. Die ganze E/A von Festplatten beruht immer auf *Block lesen und schreiben.* Ein Block ist stets 512 Byte lang. Wenn du eine 5120 Byte lange Datei „XYZ.DAT“ hast, so sind das 10 Festplattenblöcke. Zum Schreiben hast du dir in deinem

Programm einen Speicherblock „BLK“ mit 512 Byte Länge reserviert. Das war der „Puffer“. Wenn du den dann in Block 1 der Datei geschrieben hast, waren das die Bytes 1 bis 512. Der 2. Block umfasst die Bytes 513 bis 1024. Man kann auch 3 oder 4 oder 10 Blöcke puffern. Es müssen aber immer ganze Vielfache von 512 sein. Das System kann immer nur ganze Blöcke übertragen.

2. Logische und virtuelle Blöcke. Es gibt zwei Arten von Blöcken*, logische und virtuelle Blöcke.* Wenn du den 2. Block der Datei „XYZ.DAT“ schreibst, so ist das natürlich nicht wirklich Block Nr. 2 der Festplatte, sondern vielleicht der Block 4444. Denn die Datei befindet sich ja irgendwo „zufällig“ auf der Platte. Der Block 2 ist der „virtuelle“ Block in der Datei. Der Block 4444 ist der „logische“ (oder physische) Block auf der Festplatte. Im Prinzip ist es egal, ob man logischen oder virtuellen adressiert. Ein Schreiben logischer Blöcke kann allerdings schnell kriminell werden: Wenn nämlich eines kleinen Programmierfehlers versehentlich nicht Block 4444, sondern Block 5555 geschrieben wird, ist sehr schnell das gesamte Dateisystem futsch. Denn dort steht vielleicht eine völlig andere Datei. Darum muss es Sicherungssysteme geben.

3. Festplatten mounten. Mounten bedeutet *„Eingliedern der Festplatte in das Dateisystem des Betriebssystems“*. Bevor du als Anwender irgendwas mit Dateien machen duftest, musstest du die Festplatte immer mounten, am besten gleich früh beim System hochfahren mit der Datei STARTUP.CMD. Nach dem Mounten dürfen von gewöhnlichen Programmen (Tasks) nur noch virtuelle Blöcke gelesen und geschrieben werden. Ein Zugriff auf logische Blöcke ist dann verboten. Es darf also nur noch auf Block 2 der Datei zugegriffen werden. Beim Versuch eines Zugriffes auf Block 4444 der Festplatte kommt die Fehlermeldung *Privileg violation*. Somit ist sichergestellt, dass du da niemals versehentlich einen Block außerhalb deine Datei triffst.

4. Der QIO. Ein-/Ausgabe läuft in jedem Betriebssystem über Driver. Intern, aus Programmierer-Sicht, ist das über die *Betriebssystemdirektive QIO* erfolgt. Der QIO wird mit einem Makro im Assemblercode angewiesen. QIO heißt *„Queue Input/Output and Wait“,* etwa „reihe eine Ein- oder Ausgabeanforderung in die E/A-Warteschlange des Drivers ein und warte das Ende der Operation ab“. Nach dem QIO sind die Daten dann zwischen der Festplatte und dem Programmpuffer übertragen worden.

Das alles passiert „asynchron“ per DMA, d. h. unter Steuerung des Plattencontrollers, nicht durch den Prozessor oder das System. So ein QIO wird im Assemblerprogramm etwa folgendermaßen definiert:

```
DKRVB:        QIOW$C IO.RVB,2,1,,SB,,<BLK,512.>
```

DKRVB ist der Name des QIO. QIOW$C heißt das Marko. IO.RVB ist Funktionscode „Read Virtual Block“. 2 ist die logische Gerätenummer oder LUN. Die „1“ und der „SB“ sind unwichtig. BLK ist der Puffer, in den gelesen wird und 512 die Pufferlänge in Bytes.

Die eigentliche Abarbeitung unseres QIOs „DKRVB“ wird dann mit der „DIR$-Direktive“ im Laufbereich aktiviert:

```
              DIR$   #DKRVB
```

Nach Abarbeitung des Makros DIR$ stehen dann die 1024 Dateibyte im Puffer. Fundamental sind folgende vier Funktionscodes:

IO.RLB	read logical block
IO.RVB	read virtual block
IO.WLB	write logical block
IO.WVB	write virtual block

Darüber hinaus gab es aber noch eine ganze Menge anderer, nirgendwo dokumentierter Funktionscodes in dem Betriebssystem:

IO.CRE, IO.ACR, IO.ACW, IO.FNA, IO.ENA, IO.RNA, IO.DEL

Was mag das sein? CRE, konnte *„create“* heißen, DEL *„delete“*.

Geht es auch ohne FCS?

Das Verrückte war nun, auch das FCS musste nun, um an Dateien ranzukommen, den Festplattendriver ansprechen. Allerdings was das FCS auch nur „gewöhnlicher“ Assemblercode ohne irgendwelche Sonderrechte. Es gab da nicht irgendwelche Extra-Zugriffsmöglichkeiten für die Dateiarbeit. Und die einzige Möglichkeit, die das System bereitstellte, um an die Festplatte ranzukommen – war der QIO.

Wenn man nun Dateiarbeit direkt mit QIOs hinkriegte, dann konnte man sich das ganze FCS schenken. Die Programme wären dann viel einfacher, schneller und kürzer. Nur, wie rauskriegen, wie das funktioniert? Wie da herankommen?

Nun, es gab da den Betriebssystem-Debugger, das XDT. Damit konnte man das laufende Betriebssystem anhalten und den Speicher öffnen. Vom System hatten wir aber nicht allzu viel Ahnung. Es gab da eine nebulöse zentrale Geräteliste des Betriebssystems. Die begann auf der globalen Adresse $DEVHD. Bei jedem QIO musste er zunächst rauskriegen, was das für ein Gerät war. Also lag nahe, dass er da über die Adresse $DEVHD reinging.

$DEVHD hatte auf meinem System Adresse 2074. Also einen Breakpoint auf die 2074. Das war aber noch nicht allzu viel, denn auf so einem System passiert sehr viel mit Ein-/Ausgabe. Da gibt es massenhaft Unterbrechungen. Dutzendweise kamen da Interrupts von allmöglichen Geräten rein.

Genau dort blieb er nun jedesmal stehen und ich konnte den Hauptspeicher aufmachen. Wow: Ich war mit einem Mal im Systemkernel drin. Dokumentiert war da natürlich nichts. Was er da macht? Null Ahnung.

Ich habe dann ein kleines Programm mit einem ganz primitiven Dateizugriff geschrieben, etwa „öffne, ließ, und schließe eine Datei XYZ.DAT“. Alles mit FCS, wie sonst? Das in einer unendlichen Schleife und gestartet. Breakpoint auf 2074, und klatsch, schon blieb der

Debugger stehen. Jedesmal, wenn ein Befehl auf die in $DEVHD beginnende Geräteliste zugegriffen hat. Und richtig, das war bei jedem QIO der Fall.

Nun ging das Hunde Flöhen los. Du kennst das System ja nicht, weißt überhaupt nicht, wie das alles funktioniert. Den Quellcode von System und FCS hatten wir ja sowieso nicht da. Aber auch das hätte nicht viel genützt, tausende Maschinenbefehle hintereinander – viel Glück beim Debuggen.

In Assembler werden die Maschinenworte typisch in Registern gespeichert. Die Maschine hatte 8 Register, R7 war der Befehlszähler, R6 der Stackpointer. Die Register R0 bis R5 waren freie Register. Damit konntest du grundsätzlich machen, was du wolltest.

Die Register also. Als Programmierer solltest du natürlich wissen, was in deinen Registern steht. Also in „deinem Code“. Mit dem QIO ging er aber in den Kernelmodus und da wurden deine Registerinhalte gegen die Betriebssystemwerte ausgetauscht. Was mag da nun drin stehen? Es war ja der Systemdebugger und zu allem Überfluss laufen auf so einem Multi-User-System auch immer ein paar Programme zugleich auf der Maschine. Die sendeten nun bunt durcheinander Dutzende QIOs und jedesmal trapte er da rein. Das konnte dein Programm sein, aber auch irgendein anderes Programm, was parallel lief. Programme hießen in dem System Tasks. Jede Task hatte im System einen Task Control Block, TCB. Die TCB-Adresse meiner Task kannte ich. Und genau die stand bei dem Trap immer dann in Register R3, wenn das ein QIO von *meinem* Programm war. Das war schon einmal was wert.

Weiterhin war R5 war auffällig, allmählich wurde ich mit dessen Inhalt vertraut. Das war doch der Adressbereich von dem FCS in meinem Programm. Und das war ein Parameterblock von einem QIO.

Als Funktionscodes kamen da rein: IO.ACR, IO.RVB, IO.DAC. — ACR, RVB, DAC. — ACR, RVB, DAC. Immer so fort. Das waren also diese undokumentierten Herzchen. Nun konnte ich vermuten, dass er damit Dateien öffnet, liest, schließt. Mit dem ACR öffnet er, mit dem RVB liest er, mit dem DAC schließt er. Interessant. So hatte ich über die Registerinhalte von R3 und R5 den entscheidenden Schlüssel zum Verständnis gefunden.

Nun wird es richtig spannend. Unmittelbar nach dem IO.RVB kam da ein IO.RLB rein. Logisches Lesen. Und das auf meine *gemountete* Festplatte. Also das ging ja nun überhaupt nicht. Der RLB kam aber nun nicht von meiner Task. Das konnte nicht sein, denn die hätte da ja voll die *Privileg violation* abgefasst und wäre abgestürzt. In R3 stand nun allerdings nicht der TCB von meiner Task – sondern ein anderer TCB. Der verwies auf eine Task namens F11ACP.

Es gab da unter RSX diese geheimnisvollen Hilfssteuerroutinen, die ACP. Die waren etwas ganz Mysteriöses. Programme, die immer im Hintergrund liefen, von denen keiner etwas wusste, außer, dass sie „irgendwas mit Dateien machten“. Eine davon war die F11ACP. „F11ACP“ hieß etwa „File System RSX-11 Auxillary Control Processor“.

Jedenfalls hat immer nach dem RVB von meiner Task die ACP einen logischen Block gelesen. Das war möglich, denn die Hilfssteuerroutinen waren sog. „privilegierte Tasks“. Privilegierte Tasks durften auf gemountete Geräten auf logische Blöcke zugreifen. Ohne, dass die *Privileg*

violation aufgetreten wäre. Raffiniert. Was übrigens auch erklärte, wieso die Fehlermeldung *Privileg violation* hieß: Privilegverletzung, was ja eine einigermaßen nichtssagende Formulierung für einen unzulässigen Dateizugriff war. *File system access denied* wäre z. B. viel informativer gewesen. Aber so gesehen, ging *Privileg violation* schon in Ordnung.

Und natürlich war auch schon klar, dass der Festplattencontroller (also die Hardware) die physischen Blocknummern benötigt. Irgendwie musste das System die ja rausrücken.

Das war ja verrückt. Meine Task liest aus meiner Datei „XYZ.DAT“ den *virtuellen* Block 2 und unmittelbar danach liest die ACP von der Festplatte den *logischen* Block 4444. Anschließend stehe die Daten in meinem Puffer. Der RLB geht an den Festplattendriver. Die ACP hat beim System einen Zugriff auf Festplattenblock 4444 in Auftrag gegeben und das kam bei meiner Task dann als „virtueller Block 2“ an. So funktionierte das also.

So hatte ich entschlüsselt, wie unter RSX-11 das Datei lesen und schreiben intern funktionierte.

Das war aber erst die halbe Miete. Dateien müssen nämlich nicht nur gelesen und geschrieben, sondern auch *angelegt, geöffnet und gelöscht* werden.

Also wieder ein kleines Programm machen, was eine Datei „XYZ.DAT“ erzeugt, diese öffnet und dann löscht. Auch da ging es wieder mir so putzigen, nicht dokumentierten Funktionscodes los, IO.CRE, IO.ACW, IO.DEL. Wieder kamen die aus meiner Task, wie ich an R3 sah. Und auch hier hat wieder die ACP losgerattert. Aber anders. Es wurden völlig kryptische Blöcke per RLB und WLB bearbeitet. Das waren keine Blöcke aus meiner Datei mehr. Das war die zentrale Indexdatei des Dateisystems, die Hauptdatei auf der Festplatte, die das Dateisystem fundamental konstituiert hat: Die sagenumwobene INDEXF.SYS von der alle Systemprogrammierer immer nur mit höchstem Respekt sprachen. „Dort niemals irgendwas unqualifiziert rumprogrammieren“, hieß es, „damit zerschießt du dir sofort die gesamte Festplatte“. Ein bisschen verrückt war das schon. Du hast da einen IO.CRE gesendet. Die ACP hat dann was in der Indexfile rumgemacht. Und schwupp, war eine neue Datei da. Und nach dem IO.DEL gab es wieder einige Indexfile-Zugriffe von der ACP, und schwupp, war die Datei wieder verschwunden. Irgendwie schlüssig war das schon.

Nun rauskriegen, wie das genau funktioniert. Bei den QIOs waren verschiedene Parameter anzugeben. Was war deren genaue Bedeutung? Ich kannte ja Verzeichnis, Dateinamen, Dateityp, Blocknummer usw., also Parameter zum Thema „du und deine Datei“. Ich habe mir dann einfach angeguckt, wo man dem FCS dieses Zeug „vorn reingeben“ musste, und wo es dann in den QIO-Parameterblöcken „hinten rausgekommen“ ist.

Das Prinzip erkennen

Nach und nach ergab sich ein Bild. Fundamental für die Dateizugriffe waren sog. File Name Blöcke, FNBs, die in den QIOs verzeigert waren. In den FNB standen File-IDs (FID), „Dateiidentifikatoren“. Ein FID bestand immer aus zwei Zahlen, Dateinummer und Dateifolgenummer. Der FID hat eine Datei auf einem Datenträger eindeutig gekennzeichnet.

Wenn du den hattest, kamst du sofort an die Datei ran, viel unkomplizierter, als über umständliches Öffnen mit Verzeichnis, Dateinamen, Typ usw.

So begann ich das System langsam zu entschlüsseln.

Allmählich kriegte ich raus, wie man das – Tippeltappeltour – durchklempnert. Um eine Datei zu öffnen, musstest du deren FID kennen. Der steht im Verzeichnis.

Es gab damals noch nicht diese geschachtelten Verzeichnisse wie die Computer heutzutage, sondern es gab nur zwei Verzeichnisebenen, MFD und UFD. Pro Datenträger gab es ein einziges MFD (Master File Directory). In dem standen die Namen und FID der UFD (User File Directories). Ein UFD gibt es für jeden Nutzer. In dem stehen die Namen und FID seiner Dateien.

Das MFD und die UFDs waren auch nur gewöhnliche Dateien. Das MFD hatte immer den Dateinamen „000000.DIR“ und den festen FID 4/4. Mein Nutzercode war [111,114], was bedeutete, dass mein UFD den Namen „111114.DIR“ hatte.

Zum Suchen der Dateien in den Verzeichnissen gab es den QIO-Funktionscode IO.FNA (find name). So ein IO.FNA-QIO benötigt den File-ID des Verzeichnisses *in dem* er suchen soll und den Dateinamen, *nach dem* er suchen soll. Im Ergebnis liefert er dann den FID der gesuchten Datei aus dem Verzeichnis. Der FNB hatte dafür zwei Offsetadressen. Auf N.DID („Directory-ID“) steht der FID des Verzeichnisses, in dem er suchen soll und auf N.FID wird FID der gefundenen Datei eingetragen.

Es waren also zwei IO.FNA-QIOs absetzen. Mit einem ersten QIO suchst du das UFD im MFD und mit dem zweiten suchst du die Datei im UFD. Um z. B. die Datei „[111,114]XYZ.DAT“ zu finden, war zuerst in dem Verzeichnis FID 4/4 die Datei „111114.DIR“ zu suchen, als FID wird z. B. 17/2 ermittelt. Dann im Verzeichnis FID 17/2 die Datei „XYZ.DAT“ suchen. Die hat vielleicht File-ID 4711/22.

Zusätzlich muss der Vorgang noch eine logische Gerätenummer (LUN), erhalten. Anhand der LUIN wird die Geräteeinheit erkannt. Das lassen wir hier außen vor. Ein paar weitere Angaben (Attributsteuerliste ACL, Nutzerdateiattribute FDB, Statusblock SB) bleiben ebenfalls unabgehandelt.

Nun kannst du die Datei öffnen. Mit dem File-ID 4711/22, kannst du nun die Datei mit dem Funktionscode IO.ACW (access write) perfekt und schnell aufmachen. Falls nur gelesen werden soll, ist auch ein IO.ACR (access read) ausreichend.

Nun kannst du nach Herzenslust Lesen und Schreiben. Das wird nun ganz einfach mit QIOs IO.RVB bzw. IO.WVB angewiesen. Das Geniale hierbei ist, dass das voll von der Hardware gemacht wird, also vom Festplattencontroller. Das Zauberwort heißt DMA, „Direct Memory Access“. Dabei schreibt der Plattencontroller der Festplatte die Daten direkt in den Hauptspeicherbereich des Programmes rein, ohne dass es erst umständlich über Prozessor und System laufen muss. So ein Plattencontroller ärgert sich auch nicht mehr mit Dateinamen und Verzeichnis rum. Der Controller positioniert den Schreib-Lesekopf der

Festplatte und sobald Block 4444 unter ihm durchraucht, werden die Daten per DMA in deinen Puffer übertragen und fertig. Ist so ein Dateisystem nicht etwas Geniales?

Auch Datei anlegen, schließen und löschen funktioniert nach diesem Schema. Wenn du eine Datei neu anlegen willst, dann wird dies der IO.CRE tun. Der erzeugt dann einen neuen File-ID und gibt dir den im FNB zurück. Anschließend solltest du die Datei unbedingt noch ins Verzeichnis eintragen wöfür es den Funktionscode IO.ENA (enter name) gibt. — Wenn du mit der gesamten Dateiarbeit fertig bist, muss die Datei geschlossen werden. Das macht ein QIO mit Funktionscode IO.DAC (delete access). — Und mit dem Funktionscode IO.DEL kannst du Dateien löschen. Zusätzlich musst du hier auch noch den Verzeichniseintrag löschen, wöfur es den IO.RNA (remove name) gibt.

Das ist im Prinzip alles. Haben die das nicht genial programmiert?

Nun noch etwas Originalcode

Nun noch etwas Originalcode, damit man spüren kann, was es auf so einem Dampflok-Führerstand zugeht. Unsere Datei sei die [111,114]XYZ.DAT. Zunächst ist für die QIOs im Definitionsbereich des Assemblerprogramms etwa folgendes festzulegen:

```
; Zunächst 2 FNB definieren:
UNB:          .WORD  0,0,0,^R111,^R114,^R   ,^RDIR,0,0,4,4,0,0,0  ; FNB des UFD
FNB:          .WORD  0,0,0,^RXYZ,^R   ,^R   ,^RDAT,0,0,0,0,0,0,0  ; FNB der Datei
; Die QIO-Parameterblöcke:
DKFND:        QIOW$C IO.FNA,2,1,,SB,,<,,,,,UNB>             ; Find Name Directory
DKFNF:        QIOW$C IO.FNA,2,1,,SB,,<,,,,,FNB>             ; Find Name File
DKENA:        QIOW$C IO.ENA,2,1,,SB,,<,,,,,FNB>             ; Enter Name
DKRNA:        QIOW$C IO.RNA,2,1,,SB,,<,,,,,FNB>             ; Remove Name
DKCRE:        QIOW$C IO.CRE,2,1,,SB,,<FNB,ACL,100000,513.>  ; Create
DKOPE:        QIOW$C IO.ACW,2,1,,SB,,<FNB,ACL+10,100400>    ; Access Write
DKDAC:        QIOW$C IO.DAC,2,1,,SB                         ; Delete Access
DKDEL:        QIOW$C IO.DEL,2,1,,SB                         ; Delete
DKRVB:        QIOW$C IO.RVB,2,1,,SB,,<BLK,1024.>            ; Read virtual Block
DKWVB:        QIOW$C IO.WVB,2,1,,SB,,<BLK,1024.>            ; Write virtual Block
; Sonstige Daten:
ACL:          .WORD  7004,FDB,5005,FNB+N.FNAM,0             ; Attributsteuerliste
FDB:          .WORD  1,1000,0,513.,514.,0                   ; Nutzerdateiattribute,
                                                            ; 513 ist z. B. die
                                                            ; gewünschte Dateilänge
                                                            ; in Blöcken
SB:           .BLKW  2                                      ; Statusblock, 2 Worte
                                                            ; für Fehlercode
; Der Pufferbereich
BLK:          .BLKB  1024                                   ; Speicher, 1024 Byte
```

Im Laufbereich werden die QIOs dann mit der Direktive DIR$ aktiviert:

```
; Find Directory: UFD mit dem Dateinamen 111114.DIR im MFD suchen, 4,4
       DIR$   #DKFND                    ; ist der FID des MFD, der FID des UFD
                                        ; wird ermittelt und im UNB eingetragen
; Den FID des UFD aus UNB+N.FID (File-ID im UNB) nach
; FNB+N.DID (Directory-ID im FNB) kopieren:
       MOV    UNB+N.FID+0, FNB+N.DID+0  ; 1. Wort
       MOV    UNB+N.FID+2, FNB+N.DID+2  ; 2. Wort
; Find File: Datei XYZ.DAT im UFD suchen deren FID wird ermittelt und im
       DIR$   #DKFNF                    ; FNB eingetragen
       DIR$   #DKOPE                    ; Datei öffnen
```

```
DIR$   #DKRVB                    ; Virtuellen Block lesen
DIR$   #DKDAC                    ; Datei schließen
```

Und wenn es unsere Datei „XYZ.DAT“ noch gar nicht gibt und wir diese erst neu anlegen wollen? Kein Problem, das besorgt der QIO DKCRE. Anschließend ist es wichtig, dass wir auch den Dateinamen ins Verzeichnis eintragen. Das besorgt der QIO DKENA:

```
DIR$   #DKCRE                    ; Datei neu anlegen (create)
DIR$   #DKENA                    ; Datei in Verzeichnis rein (enter name)
```

Oder die Datei nicht schließen, sondern löschen? Dann an Stelle des QIO DKDAC den QIO DKDEL nehmen. Und hier auch wieder nicht vergessen, den Dateinamen auch im Verzeichnis zu löschen, da macht der QIO DKRNA:

```
DIR$   #DKDEL                    ; Datei löschen (delete)
DIR$   #DKRNA                    ; Verzeichniseintrag löschen (remove name)
```

Sieht doch sehr kryptisch aus? Na, nun mal nicht übermütig werden. Klar gibt es da auf dem Führerstand von so einer Dampflok einen Haufen Ventile, Hebel, Handräder und Manometer. So eine Dampflok ist doch keine Modelleisenbahn, die blos einen Ein-Aus-Schalter hat.

War schon ein geniales System, das olle RSX

Die DEC-Leute habe es entwickelt, aber wir haben es auch ganz gut beherrscht. Nachdem ich alles abgecodet hatte, war schnell klar, dass dieser Code viel kürzer, schneller und besser strukturiert war, als wenn man das alles über das FCS laufen lassen hätte.

Wenn du heute darüber nachdenkst, du meine Güte. Heute sitzen die in so einem ICE-Führerstand, aber wer weiß schon noch, wie das alles auch wirklich funktioniert? Wer macht sich denn da noch Gedanken, was in so einem Betriebssystem so alles passieren muss, damit dein Rechner die Daten auf der Festplatte auch findet.

Und nicht vergessen: Ein einziges falsches Byte — und die Festplatte ist kaputt. Das war früher so und heute ist das auch noch so.

Entgleisen darf der Zug nicht.

— — — — —

Abkürzungen

ACL	Attribut Control List, Attributsteuerliste. Datenstruktur für Dateiarbeit
ACP	Auxillary Control Processor. Hintergrundprogramm, das das Dateisystem organisiert
ALUN	Betriebssystemdirektive „Assign LUN“
DIR	Directory. Dateityp von UFD und MFD
DIR$	Makro zum Aktivieren von Betriebssystemdirektiven im RSX
DEC	Digital Equipment Corporation: Die haben die PDP-11 samt dem RSX-11 gebaut
DOS	Disk Operation System. Dieses Betriebssystem kam erst ganz lange nach RSX raus
DK	Disk. Bezeichnung der Festplatte im System der Robotron-Rechner
DMA	Direct Memory Access. So kommen die Daten ganz schnell vom Gerät ins Programm
FCS	File Control Services. Damit wurden unter RSX Dateizugriffe programmiert
FID	File-ID. 2 Zahlen, die eine Datei auf der Festplatte identifizieren
FDB	File Definiton Block. Nutzerdateiattribute. Datenstruktur für Dateiarbeit
FNB	File Name Block. Dateinamenblock. Fundamentale Datenstruktur für Dateiarbeit
LUN	Logische Gerätenummer für Ein-/Ausgabe. In vielen Betriebssystemen üblich
MCR	Monitor Control Routine. Die Shell des RSX
MFD	Master File Directory. Hauptverzeichnis, svw. Wurzelverzeichnis auf der Festplatte
MOV	Assemblerbefehl. Transportbefehl. Der „MOV A,B“ schafft A nach B
PDP	PDP-11. Der sagenhafte Rechner um den es hier geht. Wurde von der DEC gebaut
QIO	Queue Input Output. Betriebssystemdirektive des RSX für Ein-/Ausgabe
QIO$C	Es gibt verschiedene QIO-Formen, z. B. den QIO$, den QIO$C, den QIOW$C
RSX	RSX-11: Name des Betriebssystems der PDP-11
RM	Refresh Memory. Bezeichnung des Bildspeichers im System
UFD	User File Directory. Das Dateiverzeichnis des Anwenders, Unterverzeichnis des MFD
UIC	User Identification Code. Nutzeridentifikationscode des Anwenders
UNB	UFD Name Block. FNB (1) zum finden des UFD im MFD im Beispielcode
XDT	Exekutive-Debugger des RSX

Links

Stamerjohn, Ralph W.: f11qio.doc. Files-11 QIO Notes, April **1978.** — URL: `http://www.classiccmp.org/PDP-11/RSX-11/freeware/decus/rsx78a/346100/f11qio.doc`

Stamerjohn, Ralph W.: Up Your ACP. Version 02.00. April 17, **1980**. Monsanto. — URL: `http://www.ibiblio.org/pub/academic/computer-science/history/pdp-11/rsx/acp/acpman.doc`

Robotron: MGS K 1600, Software-Dokumentation, Betriebssystem OMOS 2.0, Anleitung für den Programmierer, Teil **3**, E/A-System. Berlin: VEB Robotron Vertrieb **1987.3.** `URL: http://9hal.ath.cx/usr/digital-ag/archiv/handbuecher/OMOS_2_0_Anleitung_fuer_den_Programmierer_Teil3_Ein-Ausgabesystem.txt`

Robotron: MGS K 1600, Software-Dokumentation, Betriebssystem OMOS 2.0, Anleitung für den Programmierer, Teil **4**, Dateiformate im Betriebssystem OMOS 1630. Berlin: VEB Robotron Vertrieb **1987.4.**

URL: http://9hal.ath.cx/usr/digital-ag/archiv/handbuecher/OMOS_2_0_Anleitung_fuer_den_Programmierer_Teil4_Dateiformate_im_Betriebssystem_OMOS.txt

Robotron: MGS K 1600, Software-Dokumentation, Betriebssystem OMOS 2.0, Anleitung für den Programmierer, Teil **5**, Dateizugriffssystem 1630 (FCS 1630). Berlin: VEB Robotron Vertrieb **1987.5.**
URL: http://9hal.ath.cx/usr/digital-ag/archiv/handbuecher/OMOS_2_0_Anleitung_fuer_den_Programmierer_Teil5_Dateizugriffssystem.txt

Die Links unterliegen nicht dem Änderungsdienst.