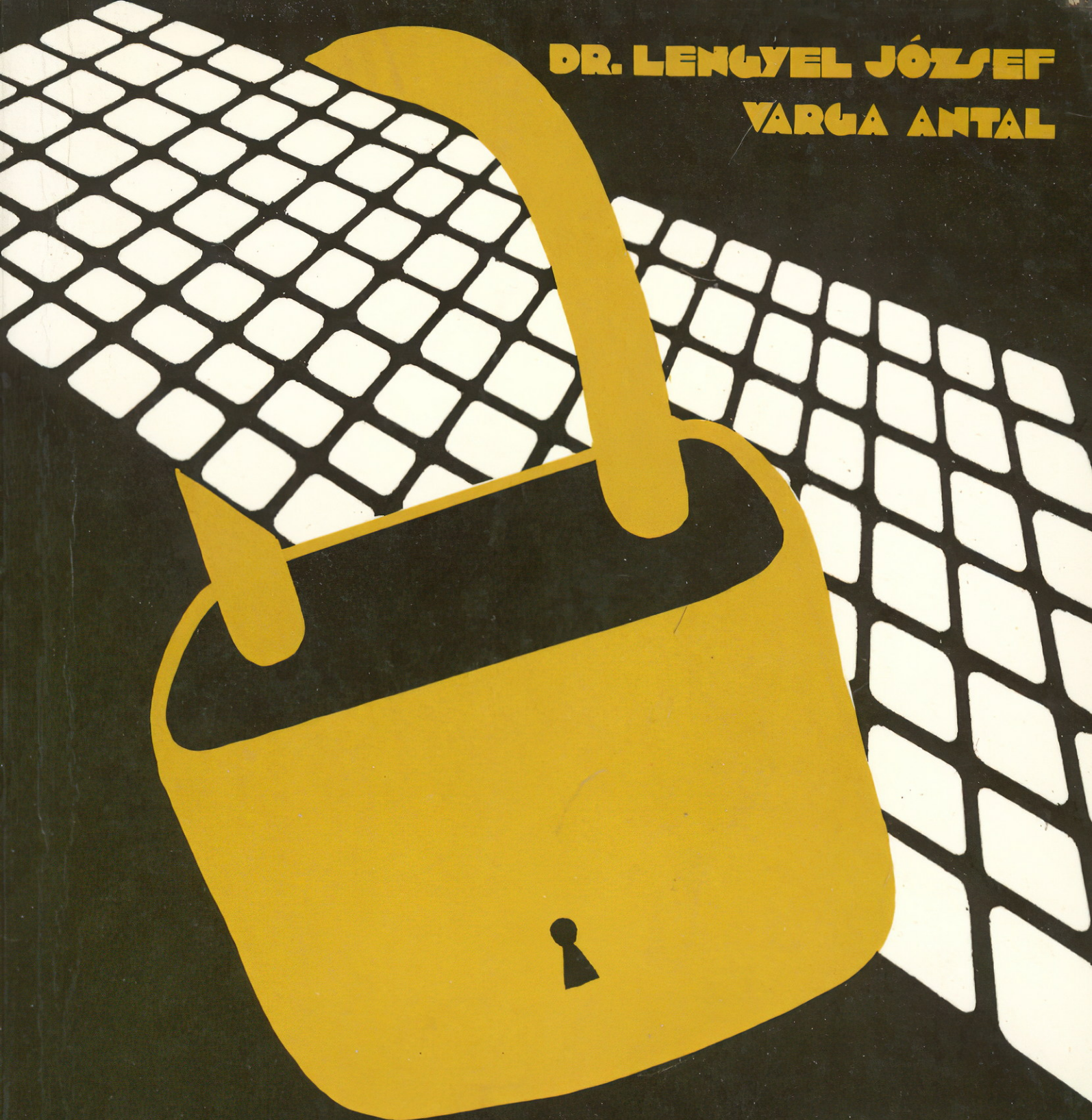
DR. LENGYEL JÓZSEF
VARGA ANTAL



LAKAT ALATT

VÉDELMI MÓDSZEREK C64-ESRE

NOVOTRADE

LENGYEL JÓZSEF – VARGA ANTAL

Lakat alatt

Novotrade Rt.

Lektorálta: Gál József

Szerkesztette: Lukács Erzsébet

A kiadásért felel RÉNYI GÁBOR, a Novotrade Rt. vezérigazgatója
Budapest, 1990

Műszaki szerkesztő: Erdősi Zoltán

Készült a Somogy Megyei Nyomdaipari Vállalat
kaposvári üzemében – 191649
Megjelent 18,2 (A/5) ív terjedelemben
Felelős vezető: Mike Ferenc igazgató

ISBN 963 585 050 6

Tartalomjegyzék

ELŐSZÓ

Könyvünkkel azoknak igyekszünk segítséget nyújtani, akik programjaikat biztonsággal szeretnék megvédeni. A címe azonban talán egy kicsit többet ígér, mint amennyit egyáltalán valóra lehet váltani. Bombabiztos védelmi módszer ui. nem létezik. Minden védelmet fel lehet törni, meg lehet fejteni, csak türelem és megfelelő szakmai ismeret szükséges hozzá. A könyvben bemutatott módszerek egy része szakmai körökben ismert, egyikkel-másikkal nyilván már az Olvasó is találkozott.

Hosszú ideig gyűjtögettük ezeket az eljárásokat, és jó részüket alkalmaztuk is. Tekintettel azonban arra, hogy a gyártó időközben többször is módosította a COMMODORE–64-es alapgép és lemezegység operációs rendszerét, elképzelhető, hogy némelyik program egyáltalán nem vagy csak kis módosítással használható.

A könyvben található alapprogramokat az Olvasó igényének megfelelően összefűzheti egy-egy új védelmi eljárás kifejlesztéséhez.

Nem volt célunk a programfeltörők munkáját elősegíteni, de a védelmi eljárások bemutatásával akarva-akaratlanul ehhez is adalékot szolgáltatunk.

1. A BASIC PROGRAMOK LISTÁZÁS ELLENI VÉDELME

Könyvünk első fejezetében arra adunk néhány ötletet, hogy hogyan lehet a BASIC program kiíratását megakadályozni, azaz a LIST parancs hatását érvényteleníteni vagy módosítani. A bemutatott módszerek természetesen egymással kombinálva is alkalmazhatók, a számtalan variációnak csak a programozó fantáziája szabhat korlátot.

Igyekeztünk az egyszerűbbektől a bonyolultabbak felé haladva az összes lehetséges listázás elleni módszerre kitérni. Ezek a módszerek azonban mind meglehetősen egyszerűek, s többé-kevésbé ismertek, ezért hatástalanításuk sem okoz különösebb problémát. Így ahol ez egyszerűen megoldható, adunk néhány tippet arra is, hogyan lehet ezeket a védelmi módszereket hatástalanítani.

1.1 Listázás elleni védelem vezérlőkarakterekkel

A listázás letiltásának legegyszerűbb módja a SHIFT-L billentyűkre (a SHIFT és az L betű együttes lenyomásával kapott karakter) épül. A megvalósítás is rendkívül egyszerű.

```
10 PRINT "EZ A PROGRAM VEDETT"
20 REM L
30 FOR I=1 TO 100
40 PRINT I
50 NEXT I
```

Ha a LIST rutin egy BASIC sorban megtalálja ezt a jelet (esetünkben a 20-as sorban), akkor a listázást megszakítja, miközben kiírja a "SYNTAX ERROR" hibaüzenetet. (Természetesen a védelem a program helyes működését nem

befolyásolja.) A módszer hiányossága, hogy a listázást csak a REM utasítás végrehajtása közben szakítja félbe, így az előző sorok és a REM utasítás még látható a képernyőn.

```
10 PRINT "EZ A PROGRAM VEDETT"
20 REM
SYNTAX ERROR
```

Ha a teljes programot szeretnénk védeni ezzel a módszerrel, akkor a SHIFT-L jelet a program első sorában kell elhelyezni.

A módszert könnyű hatástalanítani; önállóan ne is alkalmazzuk. Mivel a listázás során a REM utasítás látható, így most kiadhatjuk a

```
LIST 30-
```

parancsot, s ezzel a program további részét is megnézhetjük. Természetesen, ha minden sor elé beírjuk a SHIFT-L jelet, akkor nagyon megnehezítjük annak a dolgát, aki egy teljes képet szeretne programunkról kapni.

Van egy másik módszer, amelyikkel szintén hatástalaníthatjuk a védelmet. Ha a HELP+ segédprogram alatt kiadjuk a #L parancsot, akkor a 20-as sorban a

```
20 REM TOO MANY FILES
```

üzenet fog megjelenni, de a program további része is látható lesz. Természetesen segédprogramokkal az egyszerűbb védelmi módszerek könnyen hatástalaníthatók.

Valamivel nehezebben feltörhető védelmet jelent a következő megoldás. Helyezzünk el a BASIC programsorok között olyan sorokat, amelyek ún. mesterséges vezérlőjeleket tartalmaznak. Ezek lehetnek a közismert kurzormozgató vagy karaktertörlő vezérlőjelek, azzal a különbséggel, hogy úgy helyezzük el őket a programban, hogy beíráskor ne, kiíráskor azonban végrehajtódjanak (ezért nevezzük ezeket mesterséges vezérlőjeleknek).

Tudjuk, hogy az idézőjel (") után beírt vezérlőjeleket, pl. kurzormozgatás felfelé, a számítógép nem hajtja végre, hanem a funkciónak megfelelő vezérlőjelet tárolja a programban. Ezt használjuk fel a következő példánkban. Írjuk be a következő sort:

```
10 REM ""
```

Menjünk a második idézőjelre a kurzorral, s nyomjuk le a CTRL + 9 (RVS ON), majd a SHIFT + M billentyűket. Ekkor a második idézőjel helyett inverz mezőben egy ferde vonalat fogunk látni. Ha ezt a jelet képernyőre írjuk, pl. listázás során, akkor számítógépünk nem ezt a jelet írja ki, hanem végrehajtja a kódnak megfelelő funkciót. Esetünkben ez megfelel a RETURN billentyű lenyomásának. Írjuk be ezután a SHIFT + Q karaktert, ami megfelel a már jól ismert

"kurzor fel" jelnek. Ezek után kapcsoljuk ki az inverz írásmódot a CTRL+0 (RVS OFF) billentyűkkel, s írjunk be egy tetszőleges szöveget, pl.:

```
10 REM "..TETSZOLEGES SZOVEG
```

Ha a programot ki akarjuk listázni, akkor az első idézőjel után levő jelet a számítógép úgy értelmezi, mintha megnyomtuk volna a RETURN billentyűt, tehát a következő sor elejére áll. Ott végrehajtja a következő vezérlőjel szerinti műveletet, ami esetünkben kurzor fel, így a kurzor az előző sor elejére fog állni. Itt pedig azt a szöveget fogja kiírni, amit a vezérlőjelek után beírtunk. Esetünkben tehát a "..TETSZOLEGES SZOVEG felirat jelenik meg a képernyőn.

A módszer valamivel jobb, mint az előző, ui. a HELP+ segédprogram #L parancsa is ugyanazt az eredményt adja, mint a LIST parancs. Sornyomtatóra listázva ez semmilyen védelmet nem jelent. A módszer más vezérlőjelekkel kombinálva (pl. törlő karakterekkel) már lényegesen jobb védelmet ad.

1.2 Módosítsuk a LIST vektort!

A LIST vektor a tár $0306-os (774-es) címén található. Az operációs rendszer mindig erre a vektorra ugrik, hogy a BASIC programot olvasható szöveggé alakítsa, valahányszor a LIST parancsot kiadjuk. A vektor valójában a végrehajtandó rendszerrutin kezdőcíme, alsó (LOW) és felső byte (HIGH) alakban. Alapesetben a byte-ok tartalma $A71A (erről egy egyszerű PEEK utasítással meggyőződhetünk).

A vektort POKE utasítással vagy egy monitorprogrammal módosíthatjuk úgy, hogy a rendszer a LIST parancs végrehajtása közben egy másik rutinra ugorjon, azaz a tényleges LIST rutint átlépje. Gépeljük be a következő POKE parancsokat:

```
POKE 774,226: POKE 775,252
```

Az alsó byte most 226 ($E2), a felső byte tartalma 252 ($FC), így a LIST vektor címét a $FCE2-re (64738) módosítottuk.

Ha most kiadjuk a LIST parancsot, a rendszer listázás helyett automatikusan RESET-et hajt végre. A módszer természetesen csak akkor működik, ha a tárban van egy BASIC program, amelyik az előző módosításokat elvégzi (egyébként sem korrekt a RESET indítás).

Ezt a védelmet nem kötelező a RESET rutinra építeni. A LIST rutin kezdőcímét tetszőleges gépi kódú program kezdőcímével helyettesíthetjük, amely pl. automatikusan újraindítja vagy megsemmisíti a tárbeli BASIC programot.

1.3 Programlista helyett csak sorszámok!

Ez az egyik legjobb módszer a BASIC programok listázás elleni védelmére. Ha valaki egy ilyen típusú védelemmel ellátott BASIC programot próbál meg kilistázni, lista helyett csak sorszámokat fog látni a képernyőn. A módszer nem csekély hátránya, hogy a védelmet minden BASIC sorba be kell tenni! Gépeljük be a következő programsorokat:

```
10 :::::A = 23
20 :::::B = 85
```

Az egymást követő öt kettőspont tulajdonképpen csak a titkosításhoz szükséges karakterhelyek lefoglalására szolgál. A tényleges karaktereket monitorprogrammal kell beírni. Mielőtt erre rátérnénk, nézzük meg, hogyan néz ki ez a sor a tárban:

```
.:0800 00 0f 08 0a 00 3a 3a 3a
.:0808 3a 3a 41 b2 32 33 00 1d
.:0810 08 14 00 3a 3a 3a 3a 3a
.:0818 42 b2 38 35 00 00 00 00
.:0820 00 00 00 00 00 00 00 00
.:0828 00 00 00 00 00 00 00 00
.:0830 00 00 00 00 00 00 00 00
.:0838 00 00 00 00 00 00 00 00
.:0840 00 00 00 00 00 00 00 00
```

A $0801-es és $0802-es címen a következő BASIC sor kezdőcíme áll, alsó és felső byte alakjában. Mivel a program két sorból áll, a következő sor a $080F-es címen kezdődik. A következő két byte a BASIC sorszám, szintén alsó és felső byte alakban. Az alsó byte tartalma esetünkben 0A, a felső 00, ami megfelel a 10-es sorszámnak. Mivel a sorszám 65535-ig terjedhet, a két (alsó és felső) byte-ra mindenképpen szükség van, hiszen egy byte csak a 0-tól 255-ig terjedő sorszámok tárolására lenne elegendő. A tárban a sorszám után az öt kettőspont ASCII kódja következik. Ezek után található az A = 23 utasítás ASCII kódokkal. Ha egy érvényes BASIC utasítás lenne ezen a helyen, akkor az utasítás tokenjét találnánk itt. A következő BASIC sort az előzőtől egy $00 byte választja el. A második sor végén lévő három nulla byte a program végét jelzi.

Térjünk vissza a LIST védelemhez! Most a monitorprogram segítségével írjuk át a BASIC programba írt kettőspontokat:

```
.:0800 00 0f 08 0a 00 ff ff ff
.:0808 ff ff 41 b2 32 33 00 1d
.:0810 08 14 00 3a 3a 3a 3a 3a
.:0818 42 b2 38 35 00 00 00 00
.:0820 00 00 00 00 00 00 00 00
```

```
.:0828 00 00 00 00 00 00 00 00
.:0830 00 00 00 00 00 00 00 00
.:0838 00 00 00 00 00 00 00 00
.:0840 00 00 00 00 00 00 00 00
```

A $3A $3A $3A $3A $3A ASCII kódok helyén most $00 $FF $FF $FF $FF áll. Végezzük el ugyanezt a módosítást a 20-as sorban is! Visszatérve a monitorprogramból, adjuk ki a LIST parancsot. A képernyőn a következőt látjuk:

```
10
20
READY.
```

A sorszám megjelent, de a tényleges BASIC sor nem. Ez persze nem jelenti azt, hogy az utasítás valóban „eltűnt". Egyszerűen az történt, hogy a LIST rutin a sorszám után nulla byte-ot talált, ami számára a sor végét jelzi, és áttért a következő sorra. A módosítás a RUN parancs végrehajtását nem zavarja, a program ugyanúgy fut, mint a módosítás előtt. Ezt a védelmi módszert akkor is alkalmazhatjuk, ha egy sorban több utasítás áll. Azt azonban semmiképpen nem kerülhetjük el, hogy a védelmet minden sorba, egyenként kelljen beírnunk. Ez a munka eléggé fáradságos, így célszerű a következő segédprogramot alkalmazni:

```
100 KL=PEEK(43):REM PROGRAM KEZDETE LOW BYTE
110 KH=PEEK(44):REM PROGRAM KEZDETE HIGH BYTE
120 KC=KL+256*KH :REM PROGRAM KEZDOCIME
130 VL=PEEK(45):REM PROGRAM VEGE LOW BYTE
140 VH=PEEK(46):REM PROGRAM VEGE HIGH BYTE
150 VC=VL+256*VH :REM PROGRAM VEGE
160 FOR I=KC TO VC-3
170 FOR K=0 TO 4
180 IF PEEK(I+K)<>58 THEN 220
190 NEXT K
200 REM MEGTALALTUK MIND AZ OT KETTOSPONTOT
210 POKE I,0:I=I+4 :REM SORVEGE JEL BEIRVA
220 NEXT I
230 END
```

A programot fűzzük a védeni kívánt program mögé, majd indítsuk el a RUN (sorszám) parancs segítségével, végül futtatás után töröljük ki. A program megkeresi az öt kettősponttal megjelölt sorokat, s elhelyezi bennük a védelmet. Természetesen hosszabb programoknál a BASIC változat meglehetősen lassú, ezért érdemes ezt a részt gépi kódba átírni. A védett programot egyszerű SAVE paranccsal tárolhatjuk lemezen vagy kazettán.

Figyelem! Ha egyszer egy BASIC programba ezt a LIST védelmet beépítettük, szinte lehetetlen ismét eltávolítani! Célszerű tehát a védelem beépítése előtt az eredeti programról feltétlenül másolatot készíteni.

1.4 Programvég szimulálása

Nemcsak a LIST vektort, de a BASIC értelmezőt is megtéveszthetjük, ha tárbeli BASIC programon belül programvégét szimulálunk. Mint már az előző alfejezetből is kiderült, az értelmező a tárban a program végét három egymást követő nulla byte alapján ismeri fel. Listázás és futtatás során minden BASIC programsor feldolgozása után ellenőrzi, hogy elérte-e már a „programvég" jelet (a három nulla byte-ot), és ha igen, az aktuális feldolgozást megszakítja. A védelmi eljárást a következő kis BASIC programon mutatjuk be:

```
10 print"minta"
20 rem ....
30 goto 10
```

A módszer az értelmező leírt sajátosságán alapul. A „program vége" jelet ui. beilleszthetjük akár a program közepére is anélkül, hogy ezzel a programot tönkretennénk. Az egyetlen megkötés az, hogy a három nulla byte-ot egy BASIC sor végére kell fűzni, ahol egy nulla byte (amely a sor végét jelzi) már eleve megtalálható. A nulla byte után – amint azt az előbbiekben láttuk – a láncolási cím áll, amely a következő BASIC sor kezdetére mutat. Nézzük meg monitorprogrammal a BASIC program tárbeli elhelyezkedését:

```
.:0800 00 0e 08 0a 00 99 22 4d
.:0808 49 4e 54 41 22 00 19 08
.:0810 14 00 8f 20 2e 2e 2e 2e
.:0818 00 22 08 1e 00 89 20 31
.:0820 30 00 00 00 8f 20 2e 2e
.:0828 2e 2e 54 45 00 52 08 6e
.:0830 00 4b 48 b2 c2 28 34 34
.:0838 29 3a 8f 20 50 52 4f 47
x:0840 52 41 4d 20 4b 45 5a 44
```

A 20-as sorban a láncolási cím $0819, és ez a $080E és $080F címeken található.

Most jegyezzük fel a láncolási címet egy papírra, és írjunk a tárba helyettük nullát. Így a sor végét jelző nullával együtt megkapjuk a program végét jelző három nulla byte-ot. A láncolási címet semmiképpen nem szabad elfelejteni, hogy később a program eredeti tárképét POKE utasítással visszaállíthassuk. A következő sor kezdetét egyébként arról lehet felismerni, hogy a nullát négy nullától különböző tartalmú byte követi. Az első kettő a láncolási cím, a második kettő pedig a sorszám. Ha most a programot megpróbáljuk kilistázni, a listázás félbeszakad annál a sornál, ahol az értelmező a cím helyett nulla byte-okat talál.

Ugyanez történik, ha a programot futtatni próbáljuk. Ha most a programot lemezen tároljuk, visszatöltjük és újralistázzuk, ill. futtatjuk, az eredmény változatlan lesz. Az eredeti állapotot úgy állíthatjuk elő, hogy az előzőleg felülírt

láncolási címet POKE utasításokkal visszaírjuk. A POKE utasításokat közvetlen üzemmódban, parancsként is végrehajthatjuk, első paraméterként a láncolási cím helyét, ill. annál 1-gyel nagyobbat; második paraméterként pedig a tényleges láncolási címeket megadva. Ezután a program ismét listázható és futtatható. Még egy fontos megjegyzés: a védett programot semmiképpen nem szabad módosítani, hiszen módosítás közben a címek eltolódhatnak, és így az eredeti láncolási cím helye megváltozhat, ami a teljes BASIC programot tönkreteheti.

1.5 BASIC programok láncolása

Az előző alfejezetben már láttuk, hogy a BASIC sorok hogyan kapcsolódnak egymáshoz az ún. link byte-okon keresztül. Készíthetünk listázás elleni védelmet úgy is, hogy ezt a láncolási címet (a link byte-ok tartalmát) megváltoztatjuk. Legyen példaprogramunk a következő:

```
10 I=I+1
20 PRINTI
30 GOTO 10
```

A program tártérképe:

```
0801 0b     link byte alsó
0802 08     link byte felső
0803 0a     10 sorszám
0804 00     nullabyte
0805 49     I
0806 b2     =
0807 49     I
0808 aa     +
0809 31     1
080a 00     sor vege
080b 12     link byte alsó
080c 08     link byte felső
080d 14     20 sorszám
080e 00     nullabyte
080f 99     PRINT utasítás tokenje
0810 49     I
0811 00     sor vége
0812 1b     link byte alsó
0813 08     link byte felső
0814 1e     30 sorszám
0815 00     nullabyte
0816 89     GOTO utasítás tokenje
0817 20     SPACE
0818 31     1
0819 30     0
081a 00     a program végét jelző
081b 00     három nulla byte
081c 00
```

Ha átírjuk a $0801 cím tartalmát pl. $01-re, a LIST rutin az első sor kiírása után újból az első sort fogja kiírni, hiszen a következő sor láncolási címe $0801, s ez éppen az első sor kezdete.

Tehát a LIST rutin mindig ugyanazt a sort írja ki a képernyőre, és sohasem tér át a következő sorra. Ez a védelem a RUN parancs végrehajtását, vagyis az értelmező működését nem befolyásolja; a program változatlanul, helyesen fut.

A védett sor (sorok) mögött álló sorokat nem lehet törölni. Ha azonban az első sort töröljük, a védelem feloldódik. Éppen ezért célszerűbb ezt a védelmet úgy használni, hogy bizonyos programrészeket átugrunk, azaz a LINK byte-okat úgy módosítjuk, hogy néhány sort átugorva egy távoli programsorra mutasson. Futtatás közben az értelmező a listából kimaradt sorokat is végrehajtja, tehát a program hibátlanul fut.

Ez a védelmi módszer a legkevésbé sem feltűnő. Lehetővé teszi, hogy bizonyos fontos programsorokat (pl. azokat, amelyekben a jelszót bekérjük) egyszerűen átugorjuk, és ily módon a hívatlan szemlélő elől tökéletesen elrejtsük.

Még egy fontos tudnivaló: azokra a sorokra, amelyeket átugrunk, sosem szabad GOTO, GOSUB vagy THEN utasítással hivatkozni, mivel ezeket a sorokat a láncolási cím hiányában az értelmező sem fogja megtalálni, és ezt programmegszakítással, ill. az "UNDEF'D STATEMENT ERROR" hibaüzenettel jelzi. A BASIC program ez esetben tehát nem futhat helyesen.

Sajnos ezt a védelmet is könnyű feltörni. Például a HELP+ segédprogram #L parancsa figyelmen kívül hagyja a link byte-okat a listázás során, vagyis a teljes programot láthatóvá teszi. Éppen ezért célszerű ezt a módszert mértékkel alkalmazni, vagyis csak a kritikus programsorokat védjük le. Ha a védelem nem a program elején, hanem pl. a közepén helyezkedik el, akkor valószínűleg elkerüli az illetéktelen felhasználó figyelmét, hiszen nem gyanakszik programvédelemre.

Ugyanezen a módszeren alapszik aza gyakran alkalmazott védelem, melyben a listázás során a program ugyanazokat a sorszámokat írja ki. Módosítsuk előbbi programunkat úgy, hogy átírjuk a $080D és a $0814 rekeszek tartalmát $0A-ra. A listázás eredménye a következő lesz:

```
10 I=I+1
10 PRINT
10 GOTO 10
```

Ezzel a módszerrel megnehezíthetjük a program áttekintését, s ha még más módszerekkel kombináljuk, szinte lehetetlen a megfejtése. Még egyszer felhívjuk azonban az Olvasó figyelmét arra, hogy a vezérlésátadó utasításoknak az eredeti sorszámra kell hivatkozniuk, ellenkező esetben a program nem fog futni!

1.6 A programmódosítás megakadályozása

A programmódosítás megakadályozásának egyik legegyszerűbb módja a következő: a BASIC program végének lekérdezése. A címét a következő utasítással kiíratjuk a képernyőre:

```
PRINT PEEK(45)+PEEK(46)*256
```

Mivel ez a cím a $2D (45) és $2E (46) tárcímeken található alsó és felső byte alakban, ahhoz, hogy a BASIC programon belül hivatkozhassunk rá, át kellett alakítanunk egyetlen decimális egész számmá.

Mielőtt a program végét jelző címet betöltjük és átalakítjuk, a védeni szánt BASIC programba be kell építenünk azt a sort, amelybe a lekérdező utasítását írtuk, hiszen egyébként a program vége ismét módosulna. A kapott összeg tehát nem a helyes értéket adná.

A lekérdezés módja:

```
10 A=PEEK(45)+PEEK(46)*256
20 IF A<>00000 THEN SYS64738
```

Ha ezt a két sort beírjuk abba a programba, amelyet védeni szándékozunk, és ezután közvetlen üzemmódban lekérdezzük a program végét, megkapjuk a helyes értéket, amit a 20-as sorba kell beírnunk. Az ellenőrző összeg módosítása közben ügyeljünk arra, hogy ne írjunk egyetlen számjeggyel se többet, se kevesebbet, hiszen ez az ellenőrző összeget ismét megváltoztatná. A lekérdezést célszerű valahol a BASIC program közepén elhelyezni, hiszen így nehezebb megtalálni. Itt sem árt, ha a programba egyéb listázás elleni védelmet is beépítünk.

Ha most valaki megpróbálja a programot módosítani, pl. sort töröl a programból vagy új sort ír bele, a program a RESET rutinra ugrik, ami többek között alaphelyzetbe állítja a programkezdet- és programvégmutatókat ($2B–$2E-ig), így a program a LIST rutin számára „elvész".

A program módosítását természetesen megpróbálhatjuk úgy is, ha a védeni kívánt rész elé monitorprogram segítségével beírjuk a létező legnagyobb sorszámot ($FFFF-et). Legyen példaprogramunk a következő:

```
10 I=I+1
20 PRINTI
30 GOTO 10
40 REM VEDELEM INNEN
60 PELDAPROGRAM
70 KESZITETTE: KOVACS JOZSEF
80 KEREPESTARCSA
```

Ha monitorprogrammal a 40-es sor számát módosítjuk $FFFF-re, és újból listázzuk, a következőket látjuk:

```
10 I=I+1
20 PRINTI
30 GOTO 10
65535 REM VEDELEM INNEN
60 PELDAPROGRAM
70 KESZITETTE: KOVACS JOZSEF
80 KEREPESTARCSA
READY.
```

Az előbbi 40-es sor sorszáma most 65535 ($FFFF). Így az alatta lévő sorokat már nem érjük el, tehát sem módosítani, sem törölni nem tudjuk.

A módszer nagy hátránya, hogy a védett rész nem tartalmazhat vezérlésátadó utasításokat.

1.7 Gépi kódú program szimulálása

Kiváló listázás elleni védelmet kapunk akkor is, ha a BASIC program elejét csúsztatjuk el. A LIST parancs végrehajtása után ilyenkor a képernyőn ezt látjuk:

```
10 SYS 2210
```

Látván a SYS parancsot, az idegen szemlélő joggal feltételezi, hogy a tárbeli program gépi kódban készült. Így ha a gépi kódú programozásban nem járatos, akkor nem fogja a forrásprogramot keresni.

Nézzük meg, hogyan készíthetünk ilyen védelmet. A módszer lényege az, hogy eltoljuk a BASIC program kezdetét. Az eredeti és az új kezdőcím közötti tartományba egy kis gépi kódú programot helyezünk, amelynek feladata a BASIC új kezdetének beállítása és a RUN parancs kiadása. (Ezt a kis gépi kódú programot indítjuk a SYS utasítással.)

A védeni kívánt BASIC program legyen a következő:

```
0 REM
10 I=I+1
20 PRINTI
30 GOTO 10
```

Helyezzük el a programot a 2220-as címtől ($08AC). Így az assembler program, amelynek feladata tehát csak az, hogy az új kezdőcímet beállítsa és a BASIC programot elindítsa, a következő lehet:

```
ASS

083E                10          *=$083E
083E A9 AC          20          LDA #$AC ;AZ UJ KEZDOCIM ALSO BYTE
0840 85 2B          30          STA $2B  ;BASIC KEZDET ALSO BYTE
0842 4C 71 A8       40          JMP $A871;RUN
0845              65535         .END

ZEILEN:5   SYMBOLE:0   FEHLER:0
```

A gépi kódú program a tárban a $083E-es címen kezdődik, így a SYS 2210 paranccsal indítható. Azoknak, akik nem tudnak gépi kódban programozni, közöljük a BASIC betöltőt:

```
10 FOR I=2210 TO 2216
20 READ X:POKE I,X
30 NEXT
40 DATA 169,172,133,43,76,113,168
```

Természetesen a gépi kódú program további feladatokat is elvégezhet, pl. letilthatja a STOP, RESET billentyűket stb.
Ha a programokat elkészítettük és lemezen tároltuk, akkor a védelmi eljárás a következő:

1. Hozzuk alapállapotba a számítógépet. (Ehhez az a legjobb, ha kikapcsoljuk, majd újból bekapcsoljuk a gépet.)
2. Írjuk be a gépi kódú program kezdőcímét. Ez esetünkben:

   10 SYS 2210

3. Állítsuk át a BASIC kezdetét a 2220-as címre ($08AC).

   POKE 43,172:POKE 2219,0:NEW

   Az első POKE utasítás a BASIC program kezdetét állítja be, a második egy nulla byte-ot ír az új BASIC programterület elejére (hiszen a $0800 is ezt tartalmazza).
4. Töltsük be a gépi kódú program BASIC betöltőjét, futtassuk le, majd töröljük.
5. Töltsük be a védeni kívánt BASIC programot.
6. Írjuk vissza az eredeti BASIC kezdetet a

   POKE 43,1

   utasítással.
7. SAVE paranccsal mentsük ki a programot.

   Ha mindent pontosan végrehajtottunk, akkor a védelem biztosan sikerül. Ezek után ha valaki behívja a programunkat és kilistázza, csak a

10 SYS 2210

utasítást fogja látni, de a RUN parancsra a mögötte lévő BASIC program fog elindulni.

1.8 Hardver RESET letiltása

A RESET letiltására egyetlen védelmi módszer kínálkozik, amely a CBM80 azonosító kódra épül. Az operációs rendszer a CBM80 azonosító segítségével ellenőrzi, hogy a bővítésre (extension port) csatlakoztattunk-e modult, vagy sem.

Az operációs rendszer a CBM80 karaktersorozatot mindig a $8004 (32772) tárcímen kezeli, és ha ott nem találja, a továbbiakban nem ismeri fel. A $8000-től $8003-ig terjedő byte-ok szintén fontos információt tartalmaznak, nevezetesen azt a két címet (alsó és felső byte alakban), ahová a rendszernek a RESTORE billentyű leütése, ill. a RESET végrehajtása során ugrania kell. A $8002, $8003 címek a RESET-hez tartoznak. Azzal, hogy a byte-ok tartalmát átmásoljuk a $8000-tól $8008-ig terjedő tárterületre, egy modul aktív voltát szimulálhatjuk. Ez egyben kiváló RESET és RUN-STOP/RESTORE védelmi lehetőséget jelent.

A következő BASIC programba már beépítettük a védelmet létrehozó utasításokat.

```
5 POKE 808,254 :REM STOP BILLENTYU LETILTVA
10 C=32772
20 POKEC,195    :REM C
30 POKEC+1,194 :REM B
40 POKEC+2,205 :REM M
50 POKEC+3,56   :REM 8
60 POKEC+4,48   :REM 0
65 C=32768
66 POKE C,9
67 POKE C+1,128
70 POKE C+2,9
75 POKE C+3,128
80 FOR I=32777 TO 32777+5
82 READ X:POKEI,X:NEXT
85 DATA 32,89,166,76,174,167
100 PRINT"EBBOL NEM LEHET KILEPNI"
110 GOTO 100
READY.
```

A program az előbb ismertetett elvet követi. A 20, 30, 40, 50, 60-as sorok a $8004-től terjedő öt byte-ba beírják a CBM80-as azonosító kódot. Így a C64-es feltételezi, hogy egy bővítő modult csatlakoztattunk hozzá. A program ezután

a $8000-től beírja annak a kis gépi kódú rutinnak a kezdőcímét, amelyre a vezérlést kell adnia, ha valaki megnyomja a RESET gombot. (A RESET gomb beépítését egyénileg kell megoldani, ezt ui. a gyártó nem építi be a gépekbe!) A gépi kódú rutin a $8009-es címen kezdődik, és csak két utasításból áll. Ezeket a 80, 82, 85-ös sorok állítják elő. A védeni kívánt program tulajdonképpen a 100-as és a 110-es sorokban található. A kis gépi kódú rutin a következő két utasítást tartalmazza:

```
JSR    $A659
JMP    $A7AE
```

Az első utasítás a CHRGOT-mutatót állítja a program elejére, és egy CLR utasítást hajt végre. A második utasítás az interpreterciklusra ugrik, ami így egyenértékű a RUN parancs végrehajtásával.

Ha valaki beépíti egy BASIC programba ezt a védelmet, és a programot autostarttal indítja, a gépkezelő semmilyen módon nem tudja megszakítani a program futását, hacsak ki nem kapcsolja a gépet, vagy nem dolgozik olyan operációs rendszerben, amely egy bizonyos billentyűkombináció és a RESET egyidejű leütésére átugorja az azonosító (a CBM80) ellenőrzését.

1.9 Védekezést támogató POKE utasítás

Végezetül egy rövid összefoglalót közlünk azokról az utasításokról, amelyek hasznosak lehetnek néhány programvédelmi előírásnál.

A LIST parancsot érintő utasítások:

POKE 22,25	normál állapot
POKE 22,32	a sorszámok olvashatatlanok
POKE 22,33	! jel illesztése a sorszámokhoz
POKE 22,34	? FORMULA TO COMPLEX ERROR hibaüzenet, utána ismét normál állapot
POKE 22,35	sorszám nincs
POKE 774,226	($E2) a LIST rutin a RESET-re,
POKE 775,252	($FC) ($FCE2) = 64738-re ugrik
POKE 788,PEEK(788)+3	

A STOP billentyű letiltása:

POKE 808,254 POKE 809,255	STOP-RESTORE letiltva
POKE 818,32 POKE 819,245	SAVE letiltva
POKE 649,0	billentyűzet letiltva (RUN-STOP; RESTORE nem működik!)
POKE 649,10	billentyűzet alaphelyzetben
POKE 2048,1	RUN-ra SYNTAX ERROR (ide bármit írhatunk, akkor jelez hibát, ha tartalma nem 0)

2. EGYSZERŰ LEMEZES VÉDELMI MÓDSZEREK

Ebben a fejezetben olyan egyszerű védelmi módszereket mutatunk be, amelyekhez nem kell „komoly" programokat írni. A legtöbb esetben elegendő, ha tisztában vagyunk a lemezegység működésével, és van egy egyszerű monitorprogramunk. Védelmet ugyanis e program segítségével is létrehozhatunk. Természetesen, ha programból akarjuk majd alkalmazni, pontosan kell ismernünk a közvetlen lemezkezelés BASIC utasításait. Annak érdekében, hogy a most következő és a későbbi komolyabb példákat az Olvasó biztosan megértse és alkalmazni is tudja, röviden összefoglaljuk azokat az ismereteket, amelyekre feltétlen szükség lesz.

2.1 A diskmonitor kezelése

Ebben a fejezetben egy rendkívül egyszerű és könnyen kezelhető monitorprogram kezelését mutatjuk be. A diskmonitor, mint a neve is sejteti, olyan segédprogram, amellyel tanulmányozhatjuk a lemez szerkezetét. A program gépi kódban íródott, s a $C000 címtől kezdve helyezkedik el a memóriában. A program listája megtalálható a Novotrade kiadásában megjelent *A VC1541-es lemezegység programozása* című könyvben (264. oldaltól).

A program betöltése után a tárban marad, a $C000 címen, s innen bármikor aktivizálható a

```
SYS 49152
```

parancs segítségével. Ezután a képernyőn a következőket látjuk:

```
DISK-MONITOR V1.0
>
```

tehát a program parancsra vár.

A program által elfogadott parancsok a következők:

R: (READ) egy vektor olvasása lemezről.
Pl.: >R 12 00
: a kötelező üres hely.
12: a sáv száma hexadecimális formában
00: a szektor száma hexadecimális formában
M: képernyőn megjeleníti az előzőleg beolvasott szektor tartalmát a paraméterként megadott tartományban
Pl.: >M 00 80

ahol az M parancs után a kiírandó byte-ok kezdő- és végértéke látható (hexadecimális alakban). Erre azért van szükség, mert a teljes blokk nem fér el egyszerre a képernyőn.

```
>r 12 00
>m 80 b0
>:80  11 ff ff 01 11 ff ff 01 .▒▒..▒▒.
>:88  11 ff ff 01 11 ff ff 01 .▒▒..▒▒.
>:90  54 45 53 5a 54 2d 56 45 teszt-ve
>:98  44 45 4c 45 4d a0 a0 a0 delem
>:a0  a0 a0 5a 5a a0 32 41 a0   zz 2a
>:a8  a0 a0 a0 00 00 00 00 00    .....
>
```

(A program a lemez tartalmát hexadecimális és karakteres formában jelzi ki.)

A lemeztartalom módosítása úgy történik, hogy a kurzort (a mozgató billentyűkkel) a hexadecimális területen arra a pozícióra (byte-ra) visszük, amelyet módosítani szeretnénk. Beírjuk az új értéket hexadecimális formában, s megnyomjuk a RETURN billentyűt. A javítás eredménye azonnal megjelenik a karakteres mezőben is.

Ha a javítást befejeztük, akkor a javított szektort a memóriából kiírhatjuk lemezre a

>W (WRITE) parancs segítségével.

Pl.: >W 12 00

ahol a parancs után megadott számok a sáv–szektor értékek hexadecimális alakban.

A diskmonitor program futása közben használható az összes lemezkezelő parancs is. Mivel ezek a parancsok hasonlóak a HELP+ segédprogram lemezkezelő parancsaihoz, csak szintaxisukat ismertetjük:

>@: (RETURN)	a hibacsatorna leolvasása
>@I: (RETURN)	a lemezegység inicializálása (alaphelyzetbe hozása)
>@N: lemeznév, ID	lemez formázása
>@M:	lemez RESET

`>@V:` VALIDATE parancs a lemezre

A monitorprogramból a

`>X (RETURN)`

paranccsal térhetünk vissza a BASIC-be.

2.2 A lemez szerkezetének alapismeretei

2.2.1 A VC 1541-es BAM-ja

A BAM (Block Availability Map) feladata, hogy a szektorok foglaltságát nyilvántartsa. Tehát a BAM egy lemeztérképnek is tekinthető, amely a 18-as sáv 0-ás szektorában található, s amelyet a DOS a lemez formázása közben hoz létre.

```
>r 12 00
>m 00 ae
>:00  12 01 41 00 15 ff ff 1f  ..a..▒▒.
>:08  15 ff ff 1f 15 ff ff 1f  .▒▒..▒▒.
>:10  15 ff ff 1f 15 ff ff 1f  .▒▒..▒▒.
>:18  15 ff ff 1f 15 ff ff 1f  .▒▒..▒▒.
>:20  15 ff ff 1f 15 ff ff 1f  .▒▒..▒▒.
>:28  15 ff ff 1f 15 ff ff 1f  .▒▒..▒▒.
>:30  15 ff ff 1f 15 ff ff 1f  .▒▒..▒▒.
>:38  15 ff ff 1f 15 ff ff 1f  .▒▒..▒▒.
>:40  15 ff ff 1f 15 ff ff 1f  .▒▒..▒▒.
>:48  11 fc ff 07 13 ff ff 07  .▒▒..▒▒.
>:50  13 ff ff 07 13 ff ff 07  .▒▒..▒▒.
>:58  13 ff ff 07 13 ff ff 07  .▒▒..▒▒.
>:60  13 ff ff 07 12 ff ff 03  .▒▒..▒▒.
>:68  12 ff ff 03 12 ff ff 03  .▒▒..▒▒.
>:70  12 ff ff 03 12 ff ff 03  .▒▒..▒▒.
>:78  12 ff ff 03 11 ff ff 01  .▒▒..▒▒.
>:80  11 ff ff 01 11 ff ff 01  .▒▒..▒▒.
>:88  11 ff ff 01 11 ff ff 01  .▒▒..▒▒.
>:90  54 45 53 5a 54 2d 56 45  teszt-ve
>:98  44 45 4c 45 4d a0 a0 a0  delem
>:a0  a0 a0 5a 5a a0 32 41 a0    zz 2a
>:a8  a0 a0 a0 00 00 00 00 00     .....
>
```

Mint minden szektor, a BAM is 256 byte-ból áll. A kettőspont utáni hexadecimális szám értéke mutatja az adott sorban látható byte-ok közül az első sorszámát. Az 1. táblázat a BAM szerkezetét foglalja össze:

1. táblázat

A BAM szerkezete

Byte sorszáma		Tartalom	Megjegyzés
dec	hex	(hex)	
0/1	$0/$1	$12/$1	A tartalomjegyzék kezdetét jelzi (sáv/szektor mutató)
2	$2	$41	Az **A** karakter ASCII kódja s jelzi a DOS számára, hogy a lemezt 1541-es egységen formáztuk meg
3	$3	$0	Nem használatos
4–143	$4–$8F	–	Blokkfoglaltsági térkép (BAM)
144–159	$90–$9F	–	A lemez neve SHIFT-elt szóközzel kiegészítve
160–161	$A0–$A1	$A0	SHIFT-elt szóközjel
162–163	$A2–$A3	–	Lemez ID karaktere
164	$A4	$A0	SHIFT-elt szóköz
165–166	$A5–$A6	$32/$41	DOS verzió lemezformátum-típusa (2A)
167–170	$A7–$AA	$A0	SHIFT-elt szóköz
170–255	$AB–$FF	?	Nem használatos

2.2.2 A tartalomjegyzék szerkezete

A 18-as sávon a felhasználó nem tárolhat adatokat. Ez végig az operációs rendszer számára van fenntartva, itt található a lemez tartalomjegyzéke. Persze nem egészen olyan formában, mint azt korábban megszoktuk. Most a szokott módon töltsük be és írassuk ki a lemez tartalomjegyzékét a képernyőre:

```
load"$",8

seartching for $
loading
ready.
list

0 "tesztvedelem     " zz 2a
1     "pelda1"            prg
1     "pelda2"            prg
662 blocks free.
ready.
```

Ha ugyanezen lemez tartalomjegyzékét monitorprogrammal is megvizsgáljuk, a 18-as sáv 1-es szektorát a képernyőre íratva a következőket látjuk:

```
sys 49152

disk-monitor v1.0
>r 12 01
>m 00 40
>:00  00 ff 82 11 00 50 45 4c .✻...pel
>:08  44 41 31 a0 a0 a0 a0 a0 da1
>:10  a0 a0 a0 a0 a0 00 00 00      ...
>:18  00 00 00 00 00 00 01 00 ........
>:20  00 00 82 11 01 50 45 4c .....pel
>:28  44 41 32 a0 a0 a0 a0 a0 da2
>:30  a0 a0 a0 a0 a0 00 00 00      ...
>:38  00 00 00 00 00 00 01 00 ........
>
```

A 2. táblázatban összefoglaljuk a tartalomjegyzék-szektorok szerkezetét

2. táblázat

A tartalomjegyzék-szektorok szerkezete

Byte (hex)	Tartalom	Megjegyzés
$0	–	A következő sáv mutatója a tartalomjegyzékben
$01	–	A következő szektor mutatója a tartalomjegyzékben
$2–$1E	–	Az 1. file bejegyzése a tartalomjegyzékben
$20–$21	0	Nem használt
$22–$3F	–	2. file-bejegyzés
$40–$41	0	Nem használt
$42–$5F	–	3. file-bejegyzés
$60–$61	0	Nem használt
$62–$7F	–	4. file-bejegyzés
$80–$81	0	Nem használt
$82–$90	–	5. file-bejegyzés
$A0–$A1	0	Nem használt
$A2–$8F	–	6. file-bejegyzés
$C0–$C1	0	Nem használt
$C2–$DF	–	7. file-bejegyzés
$E0–$E1	0	Nem használt
$E2–$FF	–	8. file-bejegyzés

Egy tartalomjegyzék-szektorban 8 file-bejegyzés fér el, s mivel 18 szektorunk van, így adódik, hogy egy lemezen maximálisan

8*18 = 144

file-bejegyzés lesz. Minden szektor első két byte-ja egy mutató, amely a logikailag következő sáv és szektor számát tartalmazza. A mi példánkban ezek értéke 00FF, ami azt jelenti, hogy a tartalomjegyzék nem folytatódik más szektorokban, ez az utolsó szektora. Abban az esetben, ha a tartalomjegyzékben 8-nál több bejegyzés lenne, akkor a láncolási mutató 12 04 lenne. Azt, hogy melyik lesz a következő szektor a tartalomjegyzék esetében, a következő kis táblázatból előre megállapíthatjuk:

0 BAM
1, 4, 7, 10, 13, 16
2, 5, 8, 11, 14, 17
3, 6, 9, 12, 15, 18 *A 18-as sáv szektorainak logikai lánca.*

Tehát az 1-es szektort nem a 2-es szektor követi logikailag, hanem a 4-es, azután pedig a 7-es stb.

Hogy miért van ez így? Azért, mert a lemezegységünk 300 f/p fordulatszámmal forgatja a behelyezett lemezt. Míg a beolvasott adatok továbbítását a lemezegység hardvere, szoftvere elvégzi, s felkészül a következő szektor olvasására, addig a mechanika a lemezt kb. 2 szektorral elforgatja. Az előbbi láncolási módszert azért vezették be, hogy ne kelljen sokat várakozni arra, míg az elolvasandó szektor újra az olvasófej pozíciójához ér. Így amikor a lemezegység végzett egy szektor továbbításával, az olvasófej előtt pontosan a következő olvasandó szektor áll, tehát a várakozási idő minimális (hasonlóan magyarázható írás esetén is).

2.2.3 Közvetlen lemezkezelés parancsai

A VC1541-es DOS operációs rendszere kilenc közvetlen parancsot ismer.

U1 (Block-Read);	adatokat olvas a 1541-es RAM területére.
B-F (Buffer-Pointer);	egy tetszőleges byte-ra pozicionálja a mutatót a lemez RAM területén.
U2 (Block-Write);	adatokat ír a 1541-es RAM területéről lemezre.
M-R (Block-Read);	a 1541-es RAM vagy ROM területének beolvasása a C 64 RAM területére.
M-W (Memory-Write);	a C64 RAM területéről adatokat ír a 1541-es RAM területére.
B-A (Block-Allocate);	az aktuális szektort foglalttá teszi a lemez BAM területén.
B-F (Block-Free);	az aktuális szektort szabaddá teszi a lemez BAM területén.

M-E (Memory-Execute); egy gépi kódú rutint hajt végre a 1541-es RAM vagy ROM területén.

B-E (Block-Execute); lemezről betölti a szektor tartalmát, egy programot, a 1541-es RAM területére, s végrehajtja azt.

A parancsok használatához meg kell nyitnunk egy file-t, amelyen keresztül a parancsokat továbbítjuk a lemezegységnek. A közvetlen lemezkezeléshez szükséges OPEN utasítás alakja a következő:

```
OPEN fsz, esz, mc, "#"
```

ahol *fsz:* a logikai file-szám (1–127),
esz: a lemezegység száma (pl. 8),
mc: másodlagos cím (2–14).

Az OPEN utasítás létrehoz egy kommunikációs csatornát a C 64-es és a 1541-es lemezegység között.

Példa: OPEN 2,8,14,"#"

A példánkban megnyitottunk egy közvetlen elérésű csatornát 2-es logikai file-számmal a C 64-es oldaláról a 8-as lemezmeghajtóra, 14-es másodlagos címmel a 1541-es oldaláról.

A logikai file-szám és a másodlagos cím közötti különbség tehát a következő: a logikai file-szám azt a csatornát jelöli, amelyen a C 64-es továbbítja az író/olvasó parancsokat (GET#; INPUT#; PRINT#) a lemezegység felé. A másodlagos cím azt a csatornát jelöli, amelyen keresztül a lemezegység a közvetlen elérésű parancsokat továbbítja (U1, M-R stb.). Ez az elv tükröződik a parancsokban megadott file-számok esetében is.
Tehát pl. a lemezegység az U1 paranccsal olvas be adatokat a lemezről a 1541-es RAM területére. Innen pl. a GET# parancs segítségével olvashatjuk ki az adatokat a C 64-es RAM területére. E rövid elmélet után nézzük meg az egyes parancsok szintaxisát!

Blokk olvasása (U1) parancs

Feladata: az utasításban megadott (sáv, szektor) tartalmát beolvassa a 1541-es RAM területére.
Szintaxis:

```
PRINT#fsz,"U1";mc;m;sáv;szektor
```

ahol *fsz:* logikai file-szám (parancscsatorna),
mc: másodlagos cím,

m: 0 (értékét csak akkor kell változtatni, ha duál floppyt használunk; mivel ez hazánkban nem elterjedt, így ennek ismertetésére nem térünk ki),
sáv: sávszám (1–35),
szektor: szektorszám.

Példa:

```
100 OPEN15,8,15
110 OPEN 2,8,3,"#"
120 PRINT#15,"U1";3;0;18;0
130 INPUT#15,E1$,E2$,E3$,E4$
140 IF E1$<>"00" THEN 200
150 FOR I=0 TO 255
160 GET#2,B$
170 A=ASC(B$+CHR$(0))
180 PRINTA;
190 NEXT I
200 CLOSE2
210 PRINT:PRINT
220 PRINTE1$;E2$;E3$;E4$
230 CLOSE15
```

Ez a példaprogram beolvassa a 1541-es BAM-ját, és kiírja a byte-ok tartalmát decimálisan. Figyeljük meg a 120-as és a 160-as sorban megadott file-számok használatát! Látjuk, hogy az adatok olvasása a 1541-es oldaláról a 3-mal megadott (az OPEN utasítás másodlagos címe a 110-es sorban) csatornán történik. A 1541-es RAM-járól a C 64-es RAM-jába a 2-sel megjelölt (logikai file-szám) csatornán folyik az adatforgalom.

Blokk írása (U2) parancs

Feladata: adatokat ír a 1541-es RAM területéről a lemezre.
Szintaxis:

```
PRINT#fsz;"U2";mc;m;sáv;szektor
```

ahol az egyes paraméterek jelentése ua., mint az U1 parancs esetében.

Puffermutató beállítása (B-P)

Feladata: beállít egy mutatót (pointert) a VIC1541-es RAM területén, s a kiolvasás ettől a byte-pozíciótól kezdődhet. Értéke 0–255 lehet.
Szintaxis:

```
PRINT#fsz;"B-P";mc;bp
```

ahol *fsz:* logikai file-szám,
mc: másodlagos cím,
bp: byte-pozíció (0–255).

Példa:

```
100 OPEN15,8,15
110 OPEN 2,8,3,"#"
120 PRINT#15,"U1";3;0;18;0
130 INPUT#15,E1$,E2$,E3$,E4$
140 IF E1$<>"00" THEN 200
145 PRINT#15,"B-P";3;144
150 FOR I=1 TO 16
160 GET#2,B$
180 PRINTB$;
190 NEXT I
200 CLOSE2
210 PRINT:PRINT
220 PRINTE1$;E2$;E3$;E4$
230 CLOSE15
```

A program kiírja a lemez nevét a képernyőre. A 145-ös sorban beállítottuk a puffermutatót a 144. pozícióra, mivel a BAM-ban itt kezdődik a lemez neve. A következő GET# utasítás (160. sor) már ettől a pozíciótól olvas.

Blokkfoglalás (B-A)

Feladata: az utasításban adott blokkot a BAM-ban foglalttá nyilvánítja. A "B-W" parancs ugyan kiír egy blokkot a lemezre, de azt nem teszi foglalttá. Ezért minden közvetlen írás után nekünk kell erről gondoskodni.
Szintaxis:

PRINT#sz,"B-A";m;sáv;szektor

Példa:

```
100 OPEN15,8,15
110 OPEN 2,8,3,"#"
120 PRINT#15,"U1";3;0;18;0
130 INPUT#15,E1$,E2$,E3$,E4$
140 IF E1$<>"00" THEN 200
150 INPUT "SAV,SZEKTOR";T,S
160 PRINT#15,"B-A";0;T;S
200 CLOSE2
210 PRINT:PRINT
220 PRINTE1$;E2$;E3$;E4$
230 CLOSE15
```

Példánkban a 150-es sorban beolvasott blokkot foglaltuk le a BAM-ban. A program futtatásával a szabad blokkok számát csökkenthetjük a lemezen.

A BAM valódi tartalmát bármikor visszakaphatjuk a VALIDATE parancs segítségével:

```
OPEN 15,8,15,"V:"
CLOSE 15
```

Blokk felszabadítása (B-F)

Feladata: a BAM-ban foglaltként nyilvántartott blokkot szabaddá teszi (a B-A parancs párja).
Szintaxis:

PRINT#fsz,"B-F";m;sáv;szektor

Memória olvasása (M-R)

Feladata: a VC1541-es RAM vagy ROM területének olvasása az utasításban megadott címtől kezdve.
Szintaxis:

PRINT#fsz,"M-R"; CHR$(alsó byte) CHR#(felső byte)

Példa:

```
100 OPEN15,8,15
110 OPEN 2,8,3,"#"
130 INPUT#15,E1$,E2$,E3$,E4$
140 IF E1$<>"00" THEN 200
150 PRINT#15,"I"
160 PRINT#15,"M-R";CHR$(0)CHR$(28)
170 GET#15,B$
180 PRINTASC(B$+CHR$(0))
200 CLOSE2
210 PRINT:PRINT
220 PRINTE1$;E2$;E3$;E4$
230 CLOSE15
```

A program leolvassa az írásvédő kapcsoló állását. Az érték a $1C00 címen található. Ha az itt levő byte 4. bitje 1, akkor a lemez nincs leragasztva. Ezt egy szubrutin formájában megírva, a lemezvédelem során még felhasználhatjuk az írásvédő tapasz ellenőrzéséhez.

Memória írása (M-W)

Feladata: az utasításban megadott címtől beírja az utasításban megadott adatokat az 1541-es RAM területre.
Szintaxis:

PRINT#fsz; "M-W" CHR$(alsó byte) CHR$(felső byte) CHR$ (adatszám) adatok

ahol az adatszám az adatbyte-ok számát jelenti, értéke 1–34 lehet. Az adatokat szintén a CHR$ függvény segítségével adjuk meg.
Példa:

```
100 OPEN15,8,15
110 OPEN 2,8,3,"#"
130 INPUT#15,E1$,E2$,E3$,E4$
140 IF E1$<>"00" THEN 200
150 PRINT#15,"I"
160 PRINT#15,"M-W";CHR$(108)CHR$(2)CHR$(1)CHR$(128)
200 CLOSE2
210 PRINT:PRINT
220 PRINTE1$;E2$;E3$;E4$
230 CLOSE15
```

Példánkban a 1541-es $026C címére $80-at írtunk (160. sor), s ezzel elértük, hogy a lemezegység piros lámpája kigyulladjon, s égve is maradjon.

Programvégrehajtás (M-E)

Feladata: a 1541-es RAM vagy ROM területen levő gépi kódú program végrehajtása az utasításban megadott címtől. (Ha a RAM területen szeretnénk egy saját programot végrehajtani, azt "M-W" utasításokkal be kell írni!)
Szintaxis:

PRINT#fsz; "M-E" CHR$(alsó byte) CHR$(felső byte)

Példa:

```
10 OPEN15,8,15,"I"
20 PRINT#15,"M-W";CHR$(74);CHR$(0);CHR$(1);CHR$(85)
25 PRINT#15,"M-E";CHR$(126);CHR$(249)
30 FOR I=1 TO 100
35 PRINTI
40 PRINT#15,"M-E";CHR$(59);CHR$(250)
50 FOR K=1 TO 10 : NEXT : NEXT
60 CLOSE 15 : PRINT"O.K."
```

A példaprogram olyan állapotba hozza a lemezegységet, hogy ki-be kapcsolás után is hibás marad, nem hajlandó még a tartalomjegyzéket sem beolvasni. Így a felhasználó (aki hiába kapcsolgatja ki és be az egységet) joggal gyanakodik arra, hogy lemezegysége tönkrement. Viheti szervizbe a készüléket? Nos, erről szó sincs! Inkább nézzük meg, hogy mit csinál a program!

20-as sor: a 1541-es $4A címére $55-öt ír (itt található a fejtovábbítás lépésszáma)

25-ös sor: bekapcsoljuk a lemezegység motorját; a rutin kérdőcíme $F97E, ezt írtuk be alsó byte, felső byte formában

40-es sor: a fej továbbítását végző rutint indítottuk be. A rutin a $FA3B címen kezdődik, s egy sávval továbbítja a fejet. A sávszám felső határát a $4A címről veszi

A program tehát egy létező sávra (35-re) pozicionálja a lemezegység fejét. A fej túlmegy a még létező lemezterületen, s a mechanikus ütközők állítják meg; ahol „kiakad". Itt már nem segít a ki-be kapcsolgatás sem. A következő egynemű parancsokkal azonban „megjavíthatjuk" az egységünket:

```
OPEN 15,8,15,"U"
PRINT#15,"I"
CLOSE 15
```

Általában a lemezformázás is megteszi.

Programtöltés és -végrehajtás (B-E)

Feladata: a lemezről kitölt egy gépi kódú programot a 1541-es RAM-jába, és végrehajtja.
Szintaxis:

```
PRINT#fsz,"B-E";mc;m;szektor
```

Ezzel a lemezkezelés parancsainak ismertetését befejeztük. A következő fejezetben néhány egyszerű lemezvédelmi módszert nézünk meg.

2.3 Védelem a tartalomjegyzékben

2.3.1 Elrejtett file-név

A program védelmének egyik lehetséges módja a tartalomjegyzék célszerű átalakítása. Bizonyos módosítások a tartalomjegyzékben megakadályozzák, hogy az arra illetéktelenek a védett programállományhoz hozzáférjenek. A továbbiakban néhány példával szemléltetjük, hogyan lehet megnehezíteni, hogy valaki a tartalomjegyzéket áttekintse, ill. betöltse.

Az első példában azt mutatjuk meg, hogyan rejthetjük el egy program nevét úgy, hogy a betöltése ne legyen olyan egyszerű, mint egyébként. Gépeljük be és hajtsuk végre a következő parancsot:

```
SAVE"(SHIFT-SPACE) PROBA"; 8
```

A SHIFT-SPACE szöveg begépelése helyett természetesen a két billentyűt kell egyszerre leütni, mielőtt a file nevét beírnánk.

Az is jó megoldás, ha közvetlenül a SHIFT-SPACE mögött ismét leütjük az idézőjelet. Ha most betöltjük és kilistázzuk a tartalomjegyzéket, a következőket látjuk:

```
load"$",8
seartching for $
loading
ready.
list
0 r"tesztvedelem     " zz 2a
1    . ""proba              prg
663 blocks free.
ready.
```

Ha megpróbáljuk a programot betölteni, "MISSING FILENAME ERROR" vagy "FILE NOT FOUND" hibaüzenetet kapunk – mintha az imént bevitt állományunk nem is létezne. A magyarázat a következő: a gép az idézőjelet kezdő vagy lezáró jelnek tekinti, és így a mögé írt file-nevet nem ismeri fel. A második idézőjel a SHIFT-SPACE billentyűkkel együtt $A0 kódot jelöl, ami a tartalomjegyzékben különleges szerepet tölt be. A processzorral tehát „elhitetjük", hogy a második idézőjelhez ért. Az eredmény képtelenségnek hat: a továbbiakban a programozó nem tudja betölteni a saját programját? Szerencsére a helyzet nem ennyire súlyos. Írjuk be a LOAD parancsot ugyanúgy, ahogyan előzőleg a SAVE-et:

```
LOAD"(SHIFT-SPACE)PROBA",8
```

és az „elrejtett" program előkerül.

A tartalomjegyzék védelmére a vezérlőkarakterek is alkalmasak: a kurzormozgató billentyűk, a színbillentyűk, a DELETE és INSERT billentyűk karakterei. Ezeket a karaktereket idézőjeles módban, vagyis egy idézőjel leütése után tehetjük láthatóvá. A szövegben elhelyezett vezérlőkarakterek kiírás közben a hozzájuk rendelt művelet végrehajtását eredményezik. A LIST műveletnek ezt az apró „gyengeségét" használjuk fel védelmi célokra. Gépeljük be a következő sort:

```
SAVE""((DEL))PRO-((INS))((DEL))BA",8
```

A karakterek sorrendje: egymás után két idézőjel, majd a DEL billentyű következik, ami letörli a második idézőjelet. Ezután írjuk be a PRO- karaktersorozatot, a file-név első négy karakterét. Most nyomjuk le az INS, majd a DEL billentyűt, végül írjuk be a file-név hiányzó betűit. Az eredmény a tartalomjegyzékben:

```
0 "tesztvedelem     " zz 2a
1    "proba"             prg
663 blocks free.
```

Az Olvasónak biztosan feltűnik, hogy a file-névből egy karakter hiányzik, egy kötőjel. Ez pontosan a vezérlőkarakterek „bűne". A megszokott alakban gépelt betöltési parancs természetesen eredménytelen, azaz a mi szempontunkból éppen hogy eredményes, hiszen "FILE NOT FOUND" hibaüzenethez vezet. Ez érthető is: a gép a hiányzó kötőjel miatt nem találhatja meg a programot a tartalomjegyzékben. A betöltés csak akkor sikeres, ha a parancs alakja megegyezik a SAVE parancs alakjával.

Vigyázat! Az eredetitől eltérő operációs rendszerben elképzelhető, hogy nem érvényesek az előbb leírtak, mert a rendszer nem az eredeti LIST rutint használja (pl. SPEED-DOS).

2.3.2 Elrejtett tartalomjegyzék

A tartalomjegyzék tehát a 18-as sáv 1-es szektorán kezdődik, s a blokk első két byte-ja a láncolási címet tartalmazza. Ha ezt módosítjuk, a tartalomjegyzék kiíratását csaknem teljesen lehetetlenné tesszük. Írjunk be hexadecimális FF-et. Ha most a lemezegység kísérletet tesz a tartalomjegyzék beolvasására, az eredmény az "ILLEGAL TRACK OR SECTOR" hibaüzenet lesz, a láncolási cím ui. nem létező blokkra mutat. A módosítás a betöltést nem befolyásolja. Ha tudjuk a program nevét, akkor egy egyszerű LOAD paranccsal betölthetjük a programot a memóriába.

2.3.3 Írásvédett lemez szoftveres úton

Minden módszernek, amely a program nevével vagy a tartalomjegyzék szerkezetével végez valamilyen módosítást, van egy szépséghibája. Aki a módszert ismeri, egy monitorprogram segítségével eltávolíthatja a védelmet úgy, hogy a helyes értékeket visszaírja a lemezre. A lemezt írásvédetté tehetjük ugyan az írásvédő tapasz felhelyezésével, de ezt inkább csak a véletlen felülírások ellen használjuk, hiszen a tapaszt bárki bármikor leveheti, és módosíthatja a lemez tartalmát.

Most egy olyan írásvédelmi módszert mutatunk be, amelyet a lemezről csak a lemez újraformázásával lehet eltávolítani. (El lehet távolítani még másképpen is, pl. ha valaki jártas a lemezegység programozásában. Azonban erre csak a rutinos programfeltörők képesek. A lemezegység programozásáról a későbbiekben még szó lesz.)

A védelmi módszer lényege a következő: a BAM harmadik byte-ja egy hexadecimális 41-es értéket (az A betű kódja) tartalmaz.

```
>r 12 00
>m 00 10
>:00  12 01 41 00 15 ff ff 1f ..a..▒▒.
>:08  15 ff ff 1f 15 ff ff 1f .▒▒..▒▒.
>
```

Ez a byte jelzi a 1541-esnek, hogy a formázást a C 64-es lemezegysége végezte el. Az érték tehát kötelezően hexadecimális 41, s a NEW parancs kiadásakor (ill. befejezésekor) írja fel a DOS. Ha átírjuk ezt az értéket valamilyen más értékre, pl. hexadecimális 42-re (a B betű kódja), akkor a DOS semmilyen írási műveletet nem hajlandó elvégezni az adott lemezre, hiszen azt hiszi, hogy általa ismeretlen formátumú lemez van a meghajtóban.

```
>r 12 00
>m 00 10
>:00  12 01 42 00 15 ff ff 1f ..b..▒▒.
>:08  15 ff ff 1f 15 ff ff 1f .▒▒..▒▒.
>
```

Az olvasási műveleteket azonban nem zavarja ez az érték, így a programok betöltése zavartalan lesz. Ha ezt a módszert kombináljuk más lemezvédelmi módszerekkel, akkor már igazán megnehezítjük a programfeltörők dolgát.

2.3.4 A lemez nevének módosítása

Most nem a tartalomjegyzéket, hanem a lemez nevét módosítjuk. A lemez a $12-es sáv $00-s szektorában, a BAM-ban található. A BAM tárképét monitorprogrammal kiíratva a következőket látjuk:

```
>r 12 00
>m 80 b0
>:80  11 ff ff 01 11 ff ff 01 .▒▒..▒▒.
>:88  11 ff ff 01 11 ff ff 01 .▒▒..▒▒.
>:90  54 45 53 5a 54 2d 56 45 teszt-ve
>:98  44 45 4c 45 4d a0 a0 a0 delem
>:a0  a0 a0 5a 5a a0 32 41 a0   zz 2a
>:a8  a0 a0 a0 00 00 00 00 00    .....
>
```

A DOS a lemez nevét – a tartalomjegyzékbeli file-nevekhez hasonlóan – ASCII kódokkal tárolja (l. a 90-es sort). Az ASCII kódok mögött 16 helypótló $A0 karakter következik, majd ezután az ID és a formátumazonosító.

A védelem létrehozása nagyon egyszerű. A 90-es sor első hat byte-ját felülírjuk a "14 14 14" és a "00 00 00" konstansokkal:

```
>r 12 00
>m 80 b0
>:80  11 ff ff 01 11 ff ff 01 .▒▒..▒▒.
>:88  11 ff ff 01 11 ff ff 01 .▒▒..▒▒.
>:90  14 14 14 00 00 00 56 45 ......ve
>:98  44 45 4c 45 4d a0 a0 a0 delem
>:a0  a0 a0 5a 5a a0 32 41 a0   zz 2a
>:a8  a0 a0 a0 00 00 00 00 00    .....
>
```

Most írjuk vissza a $12 $00 blokkot a lemezre, és a monitorprogram "X" parancsával térjünk vissza BASIC-be. Inicializáljuk a lemezegységet az OPEN 1,8,15,"I" paranccsal, hogy az új BAM átkerüljön a pufferbe. Ezt a műveletet a DOS minden lemezcserénél automatikusan elvégzi, feltéve, hogy az újonnan behelyezett lemez ID-je eltér az előzőtől.

Most próbáljuk meg betölteni a tartalomjegyzéket a LOAD"$",8 parancscsal. A LIST parancs hatására (lista nélkül) READY üzenetet kapunk. A programok betöltése most is zavartalan lesz, ezt a módosítás nem befolyásolja. A rendszer a tartalomjegyzéket betöltötte ugyan a szokott módon a BASIC-tárba, de a LIST parancsot nem hajtja végre.

A „hiba" oka az általunk beírt három nulla byte. A BASIC a LIST rutin listázása közben a három nullát a program végjelének tekinti, és ha azokat elérte, a műveletet befejezi. Valójában ezen alapszik a védelmi eljárás is. A tartalomjegyzék a lemez nevével együtt benn van a BASIC-tárban, de mert a név helyén a "14 14 14 00 00 00" karakterek állnak, az értelmező a listázó parancsot nem hajtja végre.

A $14 elem nem egyéb, mint a DEL vezérlőkarakter kódja. Hatására az első három byte (0 "), amit a lemezegység processzora automatikusan felír, törlődik. A következő három nulla byte-ot az értelmező végjelnek tekinti, és így a LIST-folyamatot félbeszakítja. A képernyőn csak a READY üzenet jelenik meg. Ez a védelem is csak az eredeti operációs rendszerben működik, mert a SPEED-DOS, és más rendszerek a tartalomjegyzéket nem a BASIC-, hanem a képernyőtárba töltik!

2.3.5 A file típusának átírása

A tartalomjegyzékben egyetlen byte-ot kell megváltoztatnunk ahhoz, hogy a külső szemlélő a programot pl. soros állománynak nézze. Vegyük szemügyre ismét a tartalomjegyzék tárképét:

```
>r 12 01
>m 00 20
>:00  00 ff 82 11 00 50 52 4f .▒...pro
>:08  42 41 a0 a0 a0 a0 a0 a0 ba
>:10  a0 a0 a0 a0 a0 00 00 00       ...
>:18  00 00 00 00 00 00 01 00 ........
>
```

A $00-s sor harmadik byte-ja tartalmazza a file-típus kódját. Példánkban ez az érték $82, ami PRG típusú file-t jelöl. A DOS e kód alapján tudja eldönteni a file típusát, majd a tartalomjegyzékben a megszokott formában megjeleníteni. A megengedett file-típusok, ill. kódok:

$00 Törölt file; a tartalomjegyzékben nem látható

$80 A file törölt (DEL típus)
$81 Soros file (SEQ típus)
$82 Programfile (PRG típus)
$83 User-file (USR típus)
$84 Relatív file (REL típus)

Látható, hogy minden file-kód nyolcassal kezdődik. A kezdőérték alapján még a következő kódértelmezéseket figyelhetjük meg. $0-val kezdődő kódok a nem lezárt file-ra utalnak. A tartalomjegyzékben ezt a tényt a file-típus előtti * karakter jelzi. Tehát:

$01 *SEQ
$02 *PRG
$03 *USR
$04 a nem lezárt REL file, de ez nem fordulhat elő

Ha egy file-t felülírunk, akkor a SAVE parancsba a file neve elé egy @: karaktert kell tennünk. Felülírás közben a DOS ezt úgy jegyzi meg, hogy a file-típus kódját $A-val kezdi. (Értelemszerűen a $A4-es kód nem fordulhat elő, mert a REL file felülírás esetén is fizikailag ugyanott marad a lemezen.) Nagyon hasznos és sokszor használják is a $C-vel kezdődő file-típus kódokat. Az ilyen kódú file a törlés (SCRATCH) ellen védett, SCRATCH paranccsal nem lehet törölni a lemezről. A tartalomjegyzékben a védett file-okat a típusjelzés mögött álló < jel jelzi.

E rövid kitérő után módosítsuk a $00-s sor harmadik byte-ját pl. $81-re, ami soros file-típust jelöl. Ha a blokkot ebben a formában írjuk vissza a lemezre, a tartalomjegyzék betöltése és kiíratása után ezt látjuk:

```
load"$",8
searching for $
loading
ready.
list
0 r"tesztvedelem      " zz 2a
1     "proba"              seq
663 blocks free.
ready.
```

Ugyanezt a hatást érjük el, ha a programot a SAVE"PROGRAM,S,W",8 paranccsal tároljuk. A visszatöltéshez némi ötletre van szükség, ui. a LOAD"PROBA",8 parancs ez esetben eredménytelen. Az operációs rendszer ellenőrzi a file-típust, és ha a megadott nevű file nem program típusú, a betöltést nem hajtja végre. A

LOAD"PROBA,S,R",8

parancs azonban a kívánt eredményre vezet, hiszen közöljük a rendszerrel, hogy soros file-t kell betölteni.

Még egy ötlet: célszerű ezt a védelmet betöltő programmal együtt alkalmazni. A védeni kívánt programot, amelyet soros file-nak álcáztunk, a LOADER töltse be, miután a jelszót ellenőrizte. A módszer alkalmazását tetszőleges mértékben továbbfejleszthetjük, pl. úgy, hogy ezt a védelmet kiegészítjük a korábban elmondottak valamelyikével; megváltoztatjuk a file-nevet, elrontjuk a láncolást stb. A legjobb módszer: a program lefordítása.

A fejezet befejezéseként szeretnénk néhány további ötletet adni: a BASIC nyelven írt védőprogramot feltétlenül le kell fordítani. A lefordított program visszafejtése a legjobban felkészült „programfeltörőket" is próbára teszi. Arról sem szabad megfeledkezni, hogy a védelmi rendszer legérzékenyebb pontja általában a lemezkezelés. A program szövegében a lemezkezelő utasítások elrejtésére kell a legnagyobb gondot fordítani. A titkosítás eszközeként választhatjuk pl. a CHR$ függvényt, a visszafejtést tovább nehezítve azzal, hogy a lemezparancsok paramétereit kódoló algoritmus közbeiktatásával számítjuk ki. Mindezek után a legjobb monitorprogram sem elegendő ahhoz, hogy valaki a lefordított programban megtalálja a lemezhez címzett utasításokat.

3. PROGRAMVÉDELEMRŐL MAGAS SZINTEN

3.1 Autostart

A legtöbb komolyabb programban beépített autostart található. Annak, hogy a programozók miért fejlesztenek ki újabb és újabb autostart-technikákat programjaikhoz, igen egyszerű a magyarázata. Ha a program saját magát tölti be, az egyfajta primitív védelmet jelent a másolások ellen. A következő fejezetben erről lesz szó.

3.1.1 Mire szolgál az autostart?

Először gondoljuk végig, mire is használható az autostart. Ha az autostart lényege abban állna, hogy megkíméli a felhasználót a "RUN" parancs begépelésétől, akkor jobb lenne egyszerű rutinokat használni, amelyek kis méretük miatt rövid idő alatt betölthetők, és futásuk is igen gyors. Az igazi válasz a szoftverházak programvédelmi törekvéseiben rejlik. (A továbbiakban szükség lesz a C 64-es tárfelosztásának és assembler nyelvének alapfokú ismeretére, de az alapelvek elsajátítása után korlátlan lehetőségek tárulnak fel az olvasó előtt.) Mindig szem előtt kell tartani azt, hogy azok a programvédelmek teljesen értelmetlenek, amelyeket néhány BASIC sor kitörlésével vagy átírásával hatástalanítani lehet.

Az autostart nem egy egyszerű védelem a másolással szemben, hanem arra ad lehetőséget, hogy programunkat megvédjük azok ellen, akik jogtalanul próbálják módosítani. Természetesen ez a módszer sem tökéletes, de az ezzel ellátott programoknak nagyobb esélyük van az épen maradásra.

Ha valaki elég ügyes ahhoz, hogy különböző módokon az autostartot megkerülje, képes lehet rá, hogy programunkhoz hozzáférjen. Ehhez azonban valóban ügyes és felkészült programfeltörő szükséges, aki képes rájönni arra, hogyan lehet egy programot az eredetitől eltérő tárcímen működtetni. Ezek megvalósításához jó monitor- és diskmonitor-programra van szükség, amivel nem minden felhasználó rendelkezik. Semmiképpen nem szabad megkönnyíteni a dolgukat

azzal, hogy az általunk írt programot egy egyszerű monitorparanccsal más területre helyezve ott hibátlanul működjön. Mielőtt a bonyolult autostart-technikákkal megismerkednénk, nézzünk meg néhány egyszerűbb példát.

3.1.2 A legegyszerűbb autostart

A legegyszerűbb, ún. soros autostartot a C 64-es gyártói már eredetileg beépítették. Ehhez csak két billentyű, a SHIFT–RUN/STOP egyidejű lenyomására van szükség. Miután a két billentyűt egyszerre leütöttük, először a LOAD, majd néhány sorral alatta a PRESS PLAY ON TAPE felirat jelenik meg, és a rendszer betölti a soron következő programot (feltéve, hogy kazettás egységgel dolgozunk). Amennyiben lemezegységünk van, nagy gondot kell fordítanunk a szintaxisra.

LOAD"név",8,1
LOAD"név",8

RETURN billentyű leütése helyett nyomjuk le a SHIFT–RUN/STOP billentyűket. Ha ezeket helyesen hajtottuk végre, a program a betöltés után magától elindul, a képernyőn megjelenik a RUN parancs. Az imént ismertetett módszer nem a program sajátja, hanem csak egy betöltési mód, amely teljesen független a programtól.

3.1.3 Autostart a billentyűzetpufferből

Az előző megoldás nem valódi autostart, csak a RUN parancs begépelését szimulálja. A programnak pedig semmi – a bevezetőben ígért – előnyt nem nyújt, ráadásul minden alkalommal újra meg kell ismételni.

Próbáljunk egy végleges megoldást találni! Ilyen megoldást csak úgy kaphatunk, ha a program tartalmazza az autostart rutint, és ezzel együtt tároljuk a lemezen. Kíséreljük meg az előbbi módszert programból végrehajtani: írjuk ki a parancsot a képernyőre, és szimuláljuk a RETURN billentyű leütését. Hogy világossá tegyük az előbbieket, gépeljük be a következő sort:

```
PRINT"(CLR/HOME) PRINT5+6"
```

A parancs végrehajtása után a képernyő törlődik, és megjelenik rajta a `PRINT5+6` szöveg. Mielőtt megvilágítanánk, mi köze ennek az autostarthoz, vigyük a kurzort a fenti sor alá, és írjuk be a következőket:

```
POKE 631,19:POKE 632,13:POKE 198,2
```

Ha mindent megfelelően csinálunk, a gép most a PRINT 5+6 parancsot fogja végrehajtani úgy, mintha csak most vittük volna be a billentyűzetről. Állításunkat igazolja az is, hogy a közvetlenül bevitt parancsokhoz hasonlóan a READY üzenet 2 sorral a PRINT parancs alatt jelenik meg. Bizonyosodjunk meg erről még egyszer: figyeljük meg, hogy a POKE parancsot akárhol adtuk is ki, a READY üzenet mindig a PRINT alatt 2 sorral jelenik meg. A rendszer a POKE parancsok beadása után „úgy tett", mintha leütöttük volna a HOME, majd pedig a RETURN billentyűket. (Természetesen, ennek a három POKE parancs az oka.)

Vajon mit jelentenek az előbbi parancsokban szereplő tárcímek, és mi fog történni más, különböző paraméterekre?

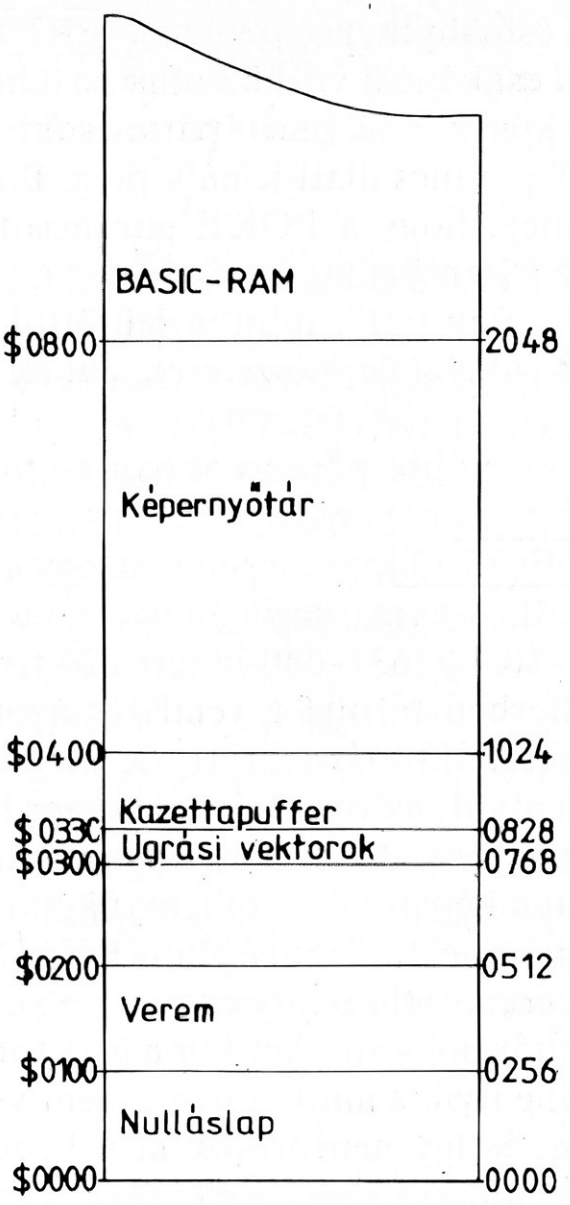
A POKE631,19:POKE632,13 parancsok nem egy közönséges tárcímre írják be az adatokat, hanem a C 64-es memóriájának egy megkülönböztetett helyére.

A C 64-esben a $0277–$0380 (631–640-ig terjedő) terület a billentyűzetpuffer helye. A billentyűzetpufferben tárolja a rendszer azoknak a billentyűknek a kódját, amelyeket a felhasználó már leütött, de még nem lettek felhasználva. Amikor leütünk egy billentyűt, az operációs rendszer beolvassa a billentyűzetpufferbe, és kiírja a képernyőre. Amikor egy programot listázunk, s közben – mialatt a programlista fut a képernyőn – billentyűket nyomunk le, a billentyűknek megfelelő jelek megjelennek a lista végén a READY üzenet után. De nem mindegyikük! Ez azért van, mert a rendszer nem folyamatosan figyeli a billentyűzetet, hanem egymás után sokszor. Amikor a lista fut, a gép ritkábban figyeli, hogy van-e lenyomott billentyű, s amit éppen „ nem vesz észre", az nem kerül be a billentyűzetpufferbe, és így nem íródik ki a képernyőre. Az első POKE paranccsal a CLR, a másodikkal pedig a RETURN leütését szimuláltuk.

A $C6 (198)-as tárcímen a billentyűzetpufferben lévő, de még ki nem írt billentyűk számát tárolja a rendszer. Az előbbi példában két billentyű leütését akartuk szimulálni, ezért került a 198-as tárcímre a 2.

Igaz ugyan, hogy a rendszer csak majdnem folyamatosan figyeli a 631–640 és a 198-as tárcímeket, ezzel a módszerrel mégsem fordulhat elő, hogy nem íródik ki a már leütött billentyű. A POKE utasításokat ui. a gép mindig végrehajtja. Az előbbi három POKE utasítás tehát egyenértékű a CLR és a RETURN gombok lenyomásával. A billentyűzetpuffer 10 karakteres hossza határt szab az ezzel a módszerrel kiírható karakterek számának; a 198-as tárcím maximális értéke csak 10 lehet.

Természetesen ezt a módszert nem parancsmódban kell majd használni, hanem be kell építeni a programba, hogy minden betöltés után sor kerülhessen a főprogram automatikus indítására. Sokakban felmerülhet az a kérdés, hogyan is nézhet ki a C 64-es alsó tárterülete. Az alsó tárfelosztást az 1. ábra mutatja.



A $0800 (2048) tárcímről kezdve tárolja a gép a BASIC programokat. Ennek a tárcímnek mindig nulla a tartalma. A program csak a $0801 (2049) tárcímtől kezdődően helyezkedik el. A C 64 tárterületeinek alsó részét, a $00–$FF (0–255) területeket nevezik nulláslapnak. Ezen a tárterületen lévő adatok különösen fontosak a rendszer számára. A C 64-es assemblere külön címzést is tartalmaz e tárterületek elérésére. A C 64-es a nulláslapon a $2B/$2C (43/44) tárcímeken tárolja a BASIC RAM kezdőcímét. A $2D/$2E (45/46) címeken található a BASIC RAM végcíme, alsó–felső byte alakban. A BASIC RAM kezdetmutatója a memóriában a következőképpen néz ki:

$002B 01 08.. .. ..

Töltsük az autostartosítani kívánt programot a tárba. Módosítsuk a BASIC terület alját úgy, hogy a 198-as címtől kezdődjön. Ezek után a billentyűzetpufferbe írjuk be a CLR és a RETURN kódjait, és a 198-as tárcímre írjunk 2-t. Ha ezzel készen vagyunk, mentsük ki a programot. (Ne feledkezzünk meg eközben a programvégmutató helyes beállításáról sem.) Olvassuk ki a 45, 46-os tárcímek tartalmát a `PRINT PEEK(45); PEEK(46)` parancsokkal. Az eljárás

tehát a következőképpen néz ki: töltsük be a BASIC programot, amelybe be szeretnénk építeni az autostartot, és gépeljük be a következő parancsot:

```
PRINT PEEK(45); PEEK(46)
```

RETURN leütése után megjelenik két szám a képernyőn, a két számot jegyezzük meg, majd töröljük a képernyőt. A képernyő legfelső sorába írjuk be a következőket:

```
POKE45,AB:POKE46,FB:RUN
```

"AB" a BASIC program végének alsó byte-ját jelöli, ami az előbb megjegyzett számok közül az első; "FB" pedig a BASIC program végének felső byte-ja, az előbbi számok közül a második. Vigyük a kurzort néhány sorral lejjebb, és írjuk be a már jól ismert POKE parancsokat:

```
POKE631,19:POKE632,13:POKE198,2:
POKE43,198:POKE44,0:SAVE"név",8
```

RETURN billentyű leütése után a gép a programot lemezre menti. A programot a `POKE 43,198` és a `POKE44,0` parancsok miatt a $C6 (198) címtől mentjük ki. A 198-as cím és a 2048 cím között olyan területet is kimentünk, amely nem tartozik a programhoz. Ide esik a képernyőtartalom is, amelynek legfelső sorában ott vannak a POKE parancsok. Mindezek miatt a program természetesen nagyobb helyet foglal el a lemezen, mint az autostart nélküli változata.

A betöltéshez hozzuk alaphelyzetbe a gépet (pl. a `POKE43,1:POKE44,8` parancsokkal vagy a gép ki-be kapcsolásával). A `LOAD"név",8,1` paranccsal tölthetjük be a programot. Az utasítás végén az 1-es nagyon fontos, nélküle már nem tudjuk betölteni a programunkat. Ezek után, ha mindent helyesen írtunk, a program automatikusan elindul. Betöltés előtt mindig ki kell adni a CLR és a NEW parancsot, különben a töltés hibás lesz.

A képernyőre a program tárolása előtt további hasznos POKE utasítást is írhatunk:

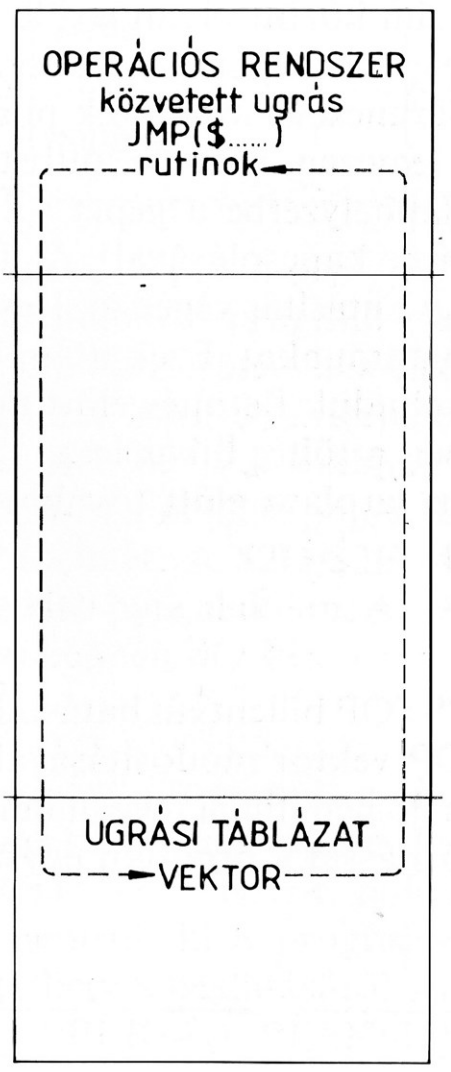
```
POKE 808,225
```

Ezzel a paranccsal a RUN/STOP billentyűt hatástalaníthatjuk. Az előbbi billentyű hatástalanítását a STOP vektor módosításával értük el. Ennek az értelme az, hogy a majdani felhasználó nem tudja megállítani a program futását. Ugyanis ezután `SAVE"név",8` utasítással a program egyszerűen másolhatóvá válna.

3.1.4 Autostart az ugrási vektorok segítségével

Az imént ismertetett módszer nagy előnye, hogy rendkívül egyszerű, nem igényel különösebb programozói ismereteket, a programozó a C 64-es tárfelosztásának behatóbb ismerete nélkül is alkalmazhatja. A módszer sok jó tulajdonság mellett számos hátránnyal is jár. Minden betöltés előtt ki kell adni egy NEW parancsot, és betöltéskor a képernyőn megjelennek a POKE parancsok (ezt pl. a színkarakterek segítségével tüntethetjük el), ráadásul a program hossza jelentősen megnő. A most következő eljárások olyan módszereket mutatnak, amelyek csökkentik az előző eljárás hátrányait. Ezek az eljárások assembler rutinokat fognak tartalmazni, de ugyanúgy, mint az előző, BASIC programok indítására szolgálnak.

A most ismertetendő eljárásokhoz tudni kell, hogy a C 64-esbe olyan ROM-ot (Read Only Memory) építettek be, melyben az operációs rendszer helyét előre rögzítették, így a gép bekapcsolás után azonnal üzemképes, nem kell egy operációs rendszert betölteni, amely – ismerve a C 64-es lassú lemezkezelését – tekintélyes időt venne igénybe. Az eljárás hátránya viszont, hogy a felhasználó akkor sem tudja megváltoztatni az operációs rendszer rutinjait, ha éppen ez a



célja. A gép tervezői megróbálták ezt a hátrányt csökkenteni. A C 64-es alsó tárterületén található az ún. ugrási táblázat, ami a rendszerrutinokhoz tartozó ugrási vektorokat tartalmazza.

Az ugrási vektorok nem a ROM-ban találhatók, így ezek könnyen megváltoztathatók. A vektorok megváltoztatásával a rendszerrutin futását megszakíthatjuk, és az általunk kívánt helyen folytathatjuk vagy módosíthatjuk, esetleg saját rutinnal cserélhetjük fel.

Az ugrási táblázat (a vektorok) a $0300-as (768) tárcímről kezdődően helyezkedik el. Minden vektor egy rutin kezdetét jelöli. A felhasználó az előbb ismertetett módon változtathatja meg őket. A 2. ábrán az ugrási táblázat és az operációs rendszer rövid, vázlatos kapcsolatát láthatjuk.

3.1.5 Az ugrási táblázat szerepe

Az ugrási táblázat tehát a vektorokat tartalmazza, a rendszer a legtöbb rutin közben használja ezeket. Az operációs rendszer akkor használ vektorokat, ha egy másik rutint hív meg. Az elágazások mindig közvetett módon, a vektorok felhasználásával történnek. A rutinok, amelyekre a vektorok mutatnak, általában közvetlenül az indirekt ugró utasítások mögött helyezkednek el. Az ugrási táblázatban szereplő első vektorok a BASIC értelmező vektorai, amelyek a mögöttük elhelyezkedő függvények címeit tartalmazzák. Ha valaki jobban el akar mélyedni az interpreter vektorok között, annak célszerű más könyvekben is, pl. *A C 64-es belső felépítése* c. könyvben utánanézni.

3. táblázat

Cím	Vektor	Leírás
$0300/0301	$E38B	A BASIC melegstart vektora. Végrehajtására az END után, ill. akkor kerül sor, ha hiba lépett fel (a hiba kódja az akkumulátorban)
$0302/#303 (770/771)	$A483	Egy sor beolvasásának vektora. A gép mindaddig nem lép ki a beolvasó/várakozó ciklusból, amíg a RETURN-t le nem ütöttük
$034/0305	$A57C	Vektor az interpreter kóddá alakításához
$0306/0307	$A71A	LIST vektor. A rendszer a programszöveg konvertálása közben átugorja
$0308/0309	$A7E4	A BASIC utasítás címének betöltése. A vektor arra a rutinra mutat (az interpreterben), amely az utasítást végrehajtja

3. táblázat folytatása

Cím	Vektor	Leírás
$0030A/030B (778/779)	$AE86	A rendszer ezt a vektort a kifejezések elemeinek kiértékelésekor hívja meg
$0311/0312 (785/786)	$B248	USR vektor; általában "ILLEGAL QUANTITY" hibaüzenet kiírására mutat
$0314/0315 (788/789	$EA31	IRQ vektor, amelyet a rendszer minden 1/60 másodpercben meghív
$0316/0317 (790/791)	$FE66	BRK vektor
$0318/0319 (792/793)	$FE47	NMI vektor; RESTORE billentyű
$031A/031B (794/796)	$F34A	OPEN vektor
$031C/031D (796/797)	$F291	CLOSE vektor
$031E/031F (798/799)	$F20E	CHKIN vektor
$0320/0321 (800/801)	$F250	CKOUT vektor
$0322/0323 (802/803)	$F333	CLRCH vektor
$0324/0325 (804/805)	$F157	INPUT vektor; általában a billentyűzetről beolvasó rutinra mutat
$0326/0327 (806/807)	$F1CA	OUPUT vektor; általában a képernyőre író rutinra mutat
$0328/0329 (808/809)	$F6ED	STOP vektor
$032A/$32B (810/811)	$F13E	GET vektor
$032C/032D (812/813)	$F32F	CLALL vektor
$032E/032F (814/815)	$FE66	Melegstart-vektor

3. táblázat folytatása

Cím	Vektor	Leírás
$0330/0331 (816/817)	$F4A5	LOAD vektor
$0332/0333 (818/819	$F5ED	SAVE vektor

A felsorolt vektorok nemcsak az autostarthoz nyújtanak segítséget, hanem néhány apró, de hasznos trükkel segítenek a programok megvédésében is.

Módosítsuk pl. a LIST vektort, hogy egy RTS-re mutasson. Ezek után a rendszer listázás helyett visszaugrik BASIC-be, azaz egyszerűen figyelmen kívül hagyja a LIST parancsot. Természetesen a LIST parancsot módosíthatjuk másképpen is. Csinálhatunk belőle egy új utasítást, vagy az RTS előtt beépíthetünk egy NEW rutint, és ezzel azok, akik megpróbálják kilistázni, automatikusan törlik is a programot. (Védelem a listázás ellen.)

Nézzünk néhány további ötletet a LIST rutin módosítására:

4. táblázat

POKE 774,	POKE 775,	Cím	Rutin
226	252	$FCE2 (64738)	RESET
68	166	$A644 (42564)	NEW
7	168	$A807 (53015)	SYNTAX ERROR
160	240	$F0A0 (61600)	A printer blokkolása

Természetesen sok minden másra is felhasználhatók a különböző vektorok, a megoldások változatosságának csak a programozó leleményessége szab határt. A SAVE rutin módosításával pl. megakadályozhatjuk, hogy valaki lemezre menthesse a programunkat. A RUN/STOP–RESTORE billentyű hatástalanítása a RESTORE vektor módosításával érhető el. Ha csak meg akarjuk szüntetni a BASIC program megszakításának lehetőségét, akkor a RESTORE hatásának megszüntetését a

```
POKE 792,193:POKE 793,254
```

utasításokkal érhetjük el. Rendelhetünk új funkciót a RUN/STOP–RESTORE billentyűkhöz úgy, hogy a vektort az általunk írt assembler rutin elejére állítjuk. (Sajnos ezt nem tehetjük meg BASIC rutinnál, mert ha ekkor megnyomjuk a RESTORE gombot, automatikusan el fog indulni az az assembler program, amire a módosított vektor mutat.)

A rendszer a RESTORE billentyű leütése után, ha a vektort még nem módosítottuk, leföldel egy bizonyos vezetéket, amely egy ún. nem maszkolható megszakítást (NMI) vált ki. A processzor ilyenkor a $FFFA címén folytatja a program futását, azaz megszakítja az éppen futó programot. Innen hívja meg az operációs rendszer az NMI rutin „testét", amely a RUN/STOP–RESTORE látható eredményét is létrehozza. Az előző két POKE parancs a $0318,$0319 (792, 793) címek, azaz az NMI vektor tartalmát módosítja $FE47-ről $FEC1-re. Ezen a helyen egy RTI (Return from Interrupt) utasítás található, ami a megszakításból való visszatérést jelenti.

Az NMI rutin egy olyan kivételes rutin, amit máshol is (nemcsak ugrási táblázatban) módosíthatunk. A C 64-esben lehetőség van külső ROM-ok csatlakoztatására és azok automatikus indítására. Ha a $8004-es (32772) címtől kezdve a CBM80 ASCII kódjait ($C3 $C2 $CD $38 $30) írjuk a memóriába, a gép úgy érzékeli, hogy egy modul van benne. A $8002/$8003-as (32770/32771) tárcímen levő vektorhoz fordul a gép minden NMI megszakítás során. A modulok úgy használják fel ezt az új NMI vektort, hogy ide a saját programjuk kezdőcímét írják, és mivel a gép bekapcsolása után a rendszer végrehajt egy NMI rutint, ez elindítja a modult. Egy kis ötlettel ezt a vektort is fel lehet használni az NMI megszakítás során: a rendszert „félrevezetjük" az előbb leírt CBM80 szöveggel, és a $8002/$8003-as (32770/32771) tárcímeket a $0318/$0319-es címekhez hasonlóan módosítjuk. De hogyan is kell ezekkel a vektorokkal autostartot előidézni? A LOAD"név",8 mindig az aktuális BASIC területre tölti a programot; a LOAD"név",8,1 parancs pedig oda tölti vissza a programot, ahonnan kimentettük. A tár egy bizonyos területét legegyszerűbben monitorprogrammal menthetjük lemezre. Akinek nincs monitorprogramja (vagy ha a MONITOR pont ott helyezkedik el a tárban, ahol a programunk van), a kimentést egy rövid rutinnal végezheti el.

A SAVE rutin gépi listája:

```
2000 LDA #$08    ;eszközszám
2002 LDX #$02    ;logikai file-szám
2004 LDY #$02    ;másodlagos cím
2006 JMP $FFBA   ;SETLFS előkészítő rutin
2009 LDA #$02    ;file-név hossza
200B LDX #$24    ;a név címének alsó byte-ja
200D LDY #$20    ;a név címének felső byte-ja
200F JMP $FFBD   ;SETNAM előkészítő rutin
2012 LDX #$00    ;a kimentendő program kezdetének alsó byte-ja
2014 LDY #$30    ;felső byte
2016 STX $D7     ;az alsó byte tárolása
2018 STY $D8     ;a felső byte tárolása
201A LDA #$D7    ;a cím helye
201C LDX #$00    ;a végcím alsó byte-ja
201E LDY #$40    ;a végcím felső byte-ja
2020 JSR $FFD8   ;SAVE rutin
2023 RTS         ;vég

2024 41 42 .. ..;a név betűinek ASCII kódja
```

A BASIC SAVE rutin listája:

```
10 AA=KEZDOCIM
20 EA=VEGCIM
30 OPEN1,8,1,"PROGRAMNEV"
40 HB=INT(AA/256):LB=AA-HB*256
50 PRINT#1,CHR$(LB);CHR$(HB);
60 FORI=AA TOEA
70 PRINT#1, CHR$(PEEK(I));
80 NEXTI
90 CLOSE1

READY.
```

Ha egy program betöltése előtt a vektorokat módosítjuk, a mutatók értéke megváltozik, és a rendszer összezavarodhat. Ha viszont csak egy vektort változtatunk és a többit alapállapotban hagyjuk kimentés előtt, majd a programot a módosított vektorral mentjük lemezre, akkor a betöltés hibátlan lesz, és a betöltött rutin is új jelentéssel fog működni.

Célszerű egy ritkán használt vektort (mint pl. a LIST); vagy egy olyan vektort választani, amelyet a gép szinte állandóan használ, mint a STOP vektor a $0328/$0329 (790/791) címen.

Mivel a rendszer a programok töltése alatt is végrehajt ugrásokat, az autostart a betöltés befejezése előtt is megtörténhet. (A szükséges rutinokat majd később ismertetjük.)

Kézenfekvőnek látszik a beolvasó/várakozó rutin használata. A rendszer ezt a rutint nagyon gyakran használja, hiszen a billentyűzetlekérdezést szinte állandóan végre kell hajtani. Állítsuk ezt a vektort a saját programunk elejére! A betöltés végén a kurzor nem jelenik meg, hanem helyette elindul a program.

Az autostartosítani kívánt programot célszerű a kazettapufferbe tenni, a $033C–$03FB (828–1019) címekre. Ha a programot ide helyezzük, akkor nincs szükség a képernyő tartalmának kimentésére, és ez jelentős helymegtakarítást jelent a lemezen.

A kazettapufferbe természetesen csak gépi kódú programokat tehetünk, és azok közül is csak a rövidebbek férnek be ide. Ha hosszabb gépi kódú programot akarunk önindítóssá tenni, akkor vagy ki kell menteni a képernyőtartalmat is, vagy egy betöltőt kell írnunk. A betöltőprogram a kazettapufferben elhelyezkedő LOAD rutin, amely elindulása után betölti és elindítja a főprogramot. Egy adott tárcímre való ugrással csak gépi kódú programot indíthatunk. A BASIC programok indítását egy rövid kapcsolórutinnal végezzük:

```
033C  JSR  $A659
033F  JMP  $A7AE
```

Az első sor adatot kér a rendszertől, hogy hol van az éppen aktuális BASIC RAM alja, ezt megadja a CHRGET rutinnak, mindezek után még egy CLR utasítást is végrehajt. A második sor végrehajtja azt a BASIC utasítást, amely-

nek kezdőcímét az iménti sor adta meg. A rendszer a második sor végrehajtása után olyan helyzetbe kerül, mintha csak egy RUN parancsot adtunk volna ki. (Ha a BASIC program autostartjával sem szeretnénk nagy helyeket foglalni a lemezen, akkor ezeket is tanácsos az említett gépi betöltőrutinnal ellátni, és a most megismert módon indítani.)

A beolvasó/várakozó rutinhoz hasonlóan a $0326/$0327 (806/807) tárcímen levő OUTPUT vektorral is elérhetünk hasonló eredményeket. Ezt a vektort is szinte állandóan használja a rendszer, hiszen minden képernyőre írásnál szükség van rá, de az OUTPUT vektornak ez a tulajdonsága gondot is okoz az autostartprogram létesítésekor. Miközben ui. kimentjük a programot, a képernyőn megjelenne a SAVING felirat, márpedig a SAVING kiírását az OUTPUT rutin végzi (ill. csak végezné). Így azonban a felirat megjelenése helyett a programunk indul el, vagyis nem tudjuk kimenteni a programunkat a módosított vektorral. Ezt a problémát is ki lehet küszöbölni.

Az egyik megoldás, hogy a programot megírjuk az eredeti helyén. Ezt követően a vektorok módosítása nélkül a memória $0300-tól a programunk végéig terjedő területet a memóriában $1000-rel odébbtoljuk. Az OUTPUT vektort az új címén a $1326/$1327-es címeken módosítjuk úgy, hogy a program kezdőcímére mutasson (természetesen a kezdőcímnek a réginek kell lennie), és a programot így mentjük lemezre. Az OUTPUT vektornak a módosítás után így kell kinézni:

$1326 3C 0C

3C az alsó, 0C pedig a felső byte lesz. Az átmásolás igen egyszerű, minden monitorprogramnak van ilyen utasítása:

T(KEZDŐCÍM) (VÉGCÍM) (ÚJ KEZDŐCÍM)

(Figyelem! Ha a kezdőcím kisebb az új kezdőcímnél, akkor a végcímnek is kisebbnek kell lennie az új kezdőcímnél.)

Hogyan helyezhetjük vissza az eredeti helyére a programot? A lemezen a program első blokkjának 2. és 3. byte-ja jelöli a program helyét a memóriában. Az iménti byte-ok közül a 3.-at kell csökkenteni 16-tal. A memóriában a programot $1000-rel toltuk feljebb, és ez a cím felső byte-jának 16-tal való megnövekedését jelenti. A csökkentést akármilyen diskmonitorral elvégezhetjük.

Sajnos ez a módszer kissé nehézkes. A következő gépi kódú lista egy olyan program listája, amely egy RUN paranccsal indítható programot autostartosít. Azoknak, akiknek nincs monitorprogramjuk, közöljük a program BASIC betöltőjét.

```
10 FORI=0TO189
20 READA$:A$=MID$(A$,1,2):B$=RIGHT$(A$,1):C$=LEFT$(A$,1)
30 B=ASC(B$)-48:IFB>9THENB=B-7
40 C=ASC(C$)-48:IFC>9THENC=C-7
50 D=16*C+B:POKE23672+I,D
60 NEXT
```

```
100 DATA A9,00    :REM C2      LDA #$00
110 DATA 8D,20,D0:REM          STA $D020
120 DATA 8D,21,D0:REM          STA $D021
130 DATA A9,05    :REM         LDA #$05
140 DATA 8D,86,02:REM          STA $0286
150 DATA A2,00    :REM         LDX #$00
160 DATA BD,94,C0:REM L1       LDA $C094,X
170 DATA C9,20    :REM         CMP #$20
180 DATA F0,07    :REM         BEQ C1
190 DATA 20,D2,FF:REM          JSR $FFD2
200 DATA E8       :REM         INX
210 DATA 4C,0F,C0:REM          JMP L1
220 DATA 20,82,C0:REM C1       JSR L2
230 DATA A2,08    :REM         LDX #$08
240 DATA A0,01    :REM         LDY #$01
250 DATA 20,BA,FF:REM          JSR $FFBA
260 DATA A2,00    :REM         LDX #$00
270 DATA A0,CF    :REM         LDY #$CF
280 DATA 86,BB    :REM         STX $BB
290 DATA 84,BC    :REM         STY $BC
300 DATA A9,00    :REM         LDA #$00
310 DATA 85,9D    :REM         STA $9D
320 DATA 20,D5,FF:REM          JSR $FFD5
330 DATA A5,90    :REM         LDA $90
340 DATA C9,40    :REM         CMP #$40
350 DATA D0,C4    :REM         BNE $C2
360 DATA A2,00    :REM         LDX #$00
370 DATA BD,95,C0:REM L3       LDA $C095,X
380 DATA F0,07    :REM         BEQ $C3
390 DATA 20,D2,FF:REM          JSR $FFD2
400 DATA E8       :REM         INX
410 DATA 4C,3E,C0:REM          JMP L3
420 DATA 20,82,C0:REM C3       JSR L2
430 DATA A2,00    :REM         LDX #$00
440 DATA A9,20    :REM         LDA #$20
450 DATA 9D,00,04:REM C4       STA $0400,X
460 DATA E8       :REM         INX
470 DATA D0,FA    :REM         BNE C4
480 DATA A2,40    :REM         LDX #$40
490 DATA A0,03    :REM         LDY #$03
500 DATA 8E,26,03:REM          STX $0326
510 DATA 8C,27,03:REM          STY $0327
520 DATA A2,00    :REM         LDX #$00
530 DATA BD,AA,C0:REM C5       LDA $C0AA,X
540 DATA 9D,40,03:REM          STA $0340,X
550 DATA E8       :REM         INX
560 DATA E0,10    :REM         CPX #$10
570 DATA D0,F5    :REM         BNE C5
580 DATA A2,00    :REM         LDX #$00
590 DATA A0,03    :REM         LDY #$03
600 DATA 86,FB    :REM         STX $FB
610 DATA 84,FC    :REM         STY $FC
620 DATA A9,FB    :REM         LDA #$FB
630 DATA A6,AE    :REM         LDX $AE
640 DATA A4,AF    :REM         LDY $AF
650 DATA 20,D8,FF:REM          JSR $FFD8
660 DATA 4C,40,03:REM          JMP $0340
670 DATA A2,00,   :REM L2      LDX #$00
680 DATA 86,B7    :REM         STX $B7
690 DATA 20,CF,FF:REM C6       JSR $FFCF
```

```
700 DATA 9D,00,CF:REM          STA $CF00,X
710 DATA E8       :REM          INX
720 DATA E6,B7    :REM          INC $B7
730 DATA C9,0D    :REM          CMP #$0D
740 DATA D0,F3    :REM          BNE C6
750 DATA 60       :REM          RTS
760 DATA 93,0D,50:REM    P
770 DATA 52,4F,47:REM  ROG
780 DATA 52,41,40:REM  RAM
790 DATA 20,4E,45:REM   NE
800 DATA 56,20,3A:REM  V :
810 DATA 20,28,55:REM   (U
820 DATA 4A,29,00:REM  J)
830 DATA 00       :REM
840 DATA A2,CA    :REM          LDX #$CA
850 DATA A0,F1    :REM          LDY #$F1
860 DATA 8E,26,03:REM          STX $0326
870 DATA 8C,27,03:REM          STY $0327
880 DATA 20,59,A6:REM          JSR $A659
890 DATA 4C,AE,A7:REM          JMP $A7AE
900 DATA 20,00,00:REM          JSR $0000
910 DATA 00       :REM          BRK

READY.
```

3.1.6 Autostart a veremből

Az autostart egy új, az eddigiektől igen távoli formája az ún. verem-autostart. A C 64-esben a verem a $0100–$01FF-ig (256–511) terjedő memóriarészen helyezkedik el (l. 1. ábra).

A C 64-esben használt verem szemléltetéséhez képzeljünk el egy teniszlabdák tárolására használatos dobozt. A labdákat egymás után helyezhetjük el a dobozban, így először az utolsóként berakott labdát vehetjük ki. Ha egy mélyebben elhelyezett labdát szeretnénk elővenni, ehhez az összes felette lévőt ki kell emelnünk a dobozból. A valóságban a példában tárgyalt doboz a ma már ismertetett memóriaterületnek, a teniszlabdák pedig számoknak felelnek meg.

A veremhez tartozik egy byte, a veremmutató. Ez a mutató jelzi a rendszer számára, hogy melyik volt a verembe utoljára helyezett byte (labda). A veremmutató mindenkori értékét hozzá kell adni a verem aljának címéhez (ami $0100), és az így kapott szám adja meg a verembe utoljára helyezett adat tárcímét. A veremmutatót gépi kódból az X regiszter segítségével írhatjuk, olvashatjuk. Az TSX utasítás a veremmutató értékét az X regiszterbe, a TXS pedig az X regiszter értékét a veremmutatóba teszi.

A C 64-es a veremben az alprogramok visszahívásához szükséges hivatkozási címeket tárolja. A gépben a programok betöltését végző rutin is egy alprogram. Érdemes elgondolkozni azon, mit eredményez az, ha a veremben lévő visszatérési cím helyett saját programunk címét írjuk. Az eredmény pontosan az, amire szükségünk volt: programunk a betöltés végeztével automatikusan elindul.

Nem sokat bonyolít az eljáráson az sem, hogy a verembe valójában nem a visszatérési címet kell beírni, hanem annál 1-gyel kisebb értékeket, alsó–felső byte alakban.

A töltőrutin, miután betöltött egy programot, megkeresi a veremben a visszatérési címet, és egyet hozzáadva, a kapott címen folytatja a program futását. Ha mi töltés közben módosítottuk ezt az értéket, a töltés után a mi programunk elejére ugrik a rendszer, és azt indítja el.

Most nézzünk néhány egyszerűbb példát a verem működésére:

```
5000 JSR $6000;
```
ugrás egy szubrutinra

```
5003 NOP;
```
akármilyen utasítás

A szubrutin meghívása a $5000-es címen történt, ezért a verembe a következő utasítás címe mínusz egy kerül, azaz $5002. Először az alsó byte ($02), másodszor pedig a felső byte ($50) kerül a verembe. A folyamat közben a veremmutató értéke 2-vel nőtt, és most $50-re mutat.

Miután a program a szubrutin végéhez ért és végrehajtott egy RTS utasítást, a rendszer (fordított sorrendben) végrehajtja az előző lépések ellentétjét, és a program futását a $5003-as címen folytatja.

A veremműködés részletes ismeretét használjuk fel egy új autostart-technika kialakítására! Betöltés közben az operációs rendszer számos rutint használ, ilyen pl. a "SEARCHING FOR" és "LOADING" üzenetek kiírása. Maga a betöltés is egy külön, önálló szubrutin, amelyet a többihez hasonlóan egy RTS utasítás zár. Az RTS utasítás végrehajtása az előzőhöz hasonlóan itt is a veremben levő utolsó címtől folytatja a rendszer futását.

A mi céljainknak az felelne meg, ha az RTS után a futás a saját programunk elején folytatódna. Természetesen sohasem tudhatjuk, hogy a veremmutató éppen hol áll. Ha az egész vermet az általunk írt program kezdőcíme mínusz eggyel töltjük fel, akkor a veremben levő összes visszatérési cím elvész, de ez teljesen lényegtelen. A rendszer mindenütt a mi programunk kezdőcímét fogja találni, és az első RTS utasításnál a program futását a mi programunk elején folytatja. Ezzel az eljárással sikerült a veremmutatót megkerülnünk. Már csak az a kérdés maradt hátra, hogy hol helyezzük el a programunkat, azaz milyen értékkel töltsük fel a vermet. Célszerű lenne, ha az alsó és felső byte-ok értéke ugyanaz lenne, például $02. Az RTS-ről elmondottak szerint ez azt jelenti, hogy a programnak $0202+$01 = $0203 (514+1 = 515) címen kell kezdődnie. A $0203 elhelyezkedő tárterületet a BASIC beviteli puffer foglalja el. A BASIC beviteli puffer $0200-től $0258-ig (512–600) tart, a betöltés folyamán semmire sem használjuk, ezért programunkat nyugodtan írhatjuk erre a területre. Ez a terület sajnos csak 88 karakter hosszú, ezért hosszabb programok elhelyezésére nemigen alkalmas. Erre a területre ezért csak a már említett betöltő programokat tehetjük. Mivel a vermet nem tudjuk az eredeti helyen módosítani, hiszen a különböző rutinok folyamatosan használják, ezért célszerű, ha a vermet és az utána következő programot $1100-tól kezdve írjuk meg a memóriában. Ezen a címen mentsük ki, és mint a 3.1.5 alfejezet végén, tegyük a helyére.

A most következő program egy tetszőleges programba épít be az előbb ismertetett autostart-indítást. Az első program $C000-től (49152) kezdődik, és feladata, hogy $1100-tól $11FF-ig terjedő területet $02-vel töltse fel.

```
C000 LDA #$02      ;az érték betöltése
C002 LDX #$00      ;mutató nullázása
C004 STA $1100,X   ;az érték tárolása
C007 INX           ;a mutató növelése
C008 BNE $C004     ;ha < 00 (ff+1) ,következő érték
C00A RTS           ;visszaugrás
```

A tényleges autostart-generáló program:

```
1200 BRK          ;három nullbyte
1201 BRK          ;
1202 BRK          ;
1203 LDX #$08     ;a floppy száma
1205 LDY #$01     ;másodlagos cim
1207 JSR $FFBA    ;file paraméter beállitása
120A LDX #$1B     ;a file-név cím alsó byte-ja
120C LDY #$12     ;            felső byte-ja
120E LDA #$04     ;a file-név hossza
1210 JRS $FFBD    ;a név parameter beallitasa
1213 LDA #$00     ;a flag átkapcsolasa a LOAD-hoz
1215 JSR $FFD5    ;a file betöltése
1218 JMP $8000    ;kezdőcímre ugrás

121B 4E 45 56 00 00 00 00
```

A jobb megértés kedvéért írtunk még egy rutint a $8000 címtől kezdve, amely a keret színét ciklikusan változtatja.

```
8000 LDX #$09     ;az első mutató beállítása
8002 LDY #$00     ;a második mutató beállítása
8004 INC $D020    ;keretszín növelése
8007 INX          ;első mutató növelése
8009 BNE $8004    ;ha < 255, tovább
800A INY          ;második mutató növelese
800B BNE $8004    ;ha < 255, tovább
800D RTS          ;visszaugrás
```

3.1.7 Autostart betöltés közben

Az eddigi módszerek mindegyike olyan vektort módosított, amelyet a program betöltése után használt a rendszer. A módosított vektor eredeti feladatát „elfelejtve" a mi programunkat indította. Például az OUTPUT vektor a READY felirat kiírása helyett a RUN szerepét játszotta. Logikusnak látszik az a megállapítás, hogy a töltés közbeni programindításhoz olyan vektort kell találnunk, amelyet nemcsak a betöltés befejezése után használ a rendszer. E vektor átírásával elérhetjük, hogy a töltés közben a mi programunk vegye kezébe az irányítást.

Az eljárás nehézsége éppen abból a sajátosságból adódik, hogy az autostart még a teljes program betöltése előtt megtörténik. Az a program nem vezérelhet, amelyik még nincs betöltve. A másik nehézség az, hogy a program nem helyezkedhet el akárhol.

A kijelölt feladat elvégzését a STOP vektorral oldhatjuk meg, amely a $0328/$0329-es (808/809) címeken helyezkedik el. A STOP rutint a rendszer minden töltés során nagyon sokszor meghívja. A STOP rutin feladata a RUN/STOP billentyű állandó figyelése, és a gomb megnyomása esetén a töltés vagy a BASIC program futásának felfüggesztése. A teljes igazság az, hogy a C 64-es minden byte beolvasása után meghívja a STOP rutint...

Az előző példákban azt használtuk ki, hogy a gép a programokat oda tölti, ahová mi akarjuk, s így a vektoroknak új értéket adhatunk. A STOP vektorral más a helyzet. A memóriát a C 64 alulról kezdi feltölteni, és ezért először az alsó byte-ot módosíthatjuk. Az alsó byte módosításával csak a régi STOP rutin $FF byte-os körzetébe ugorhatunk, ez viszont a ROM-ban helyezkedik el, ezért az ugrás zavarokat okozhat. Mindebből már következik, hogy nem szabad megváltoztatnunk a vektor felső byte-ját, csak az alsót. A rendszer most az új alsó és a régi felső byte-ból új címet képez, és ezt tekinti a STOP rutin kezdőcímének.

Ha saját hasznunkra akarjuk felhasználni ezt a rutint, akkor most nem a vektort kell a rutin kezdőcíméhez igazítani, hanem a kezdőcímet a vektorhoz. Mivel a betöltés alulról indul, a STOP vektor előtti memóriacímre már tölthetünk működőképes programot. A STOP vektor eredeti alsó byte-ja $ED, ami azt jelenti, hogy a mi programunk három helyen kezdődhet, a

$00ED; $01ED; $02ED

címeken. Ezek közül csak az utolsó, a $02ED használható. A $02A8-tól $02FF-ig terjedő területet egyik rutin sem használja a töltés folyamán. Ez a terület közel helyezkedik el a STOP vektorhoz, így nem kell nagy területet fölöslegesen lemezre menteni.

Azzal, hogy amikor a kezünkbe vesszük a vezérlést, nincs még az egész program a tárban, lehetőségünk van a gép eredeti töltőrutinját a sajátunkra cserélni. A rendelkezésünkre álló hely nagyon kicsi: $02ED-től $02FF-ig csak 18 byte; igaz, hogy ezt még a $02A8–$02ED-ig terjedő területtel kibővíthetjük. A felesleges memóriák betöltésének elkerülése végett célszerű a töltőrutint a kazettapuffer végének betöltése után megszakítani. A kazettapufferbe már be lehet tenni egy tömören megírt töltőrutint, esetleg csak azt, amelyik a tényleges töltőrutint – ami a memóriában már akárhol lehet – betölti.

Azt, hogy a betöltés elérte-e már a $03FF-es tárcímet, legegyszerűbben egy kódsorozattal vizsgálhatjuk, amelyet a kazettapuffer végén helyeztünk el.

Mire jó a saját töltőprogram?

Ezt az eljárást felhasználhatjuk arra, hogy a lemezen egyetlen file-ban tároljunk különböző tárcímeken elhelyezkedő programokat. A betöltő rutin pl. egy

kóddal érzékelheti, hogy a betöltést nem folyamatosan kell végezni a memóriában, és az utána következő címmel azt is megadhatjuk, hogy hol folytassa a betöltést. Ezzel a kódos rendszerrel a betöltő rutint elég csak egyszer megírni, nincs szükség mindig új és új írására.

Az eddigiek összefoglalásaképpen először írjunk egy programot a $02A8–$02FF-ig. (Vigyázzunk arra, hogy a program a $02ED címtől induljon!) A programot folytathatjuk a kazettapufferben is. Mentsük ki a programot a $02A8-tól a kazettapuffer végéig, majd lemezen módosítsuk a STOP vektor alsó byte-ját úgy, hogy a $02ED címre mutasson. Az így elkészült rutinhoz a rutintól függően mentsük hozzá a programokat. A hozzámentendő programot a kódokkal fejeljük meg.

Hasznos lehet az a trükk, hogy az általunk írt rutinnal elérhetjük, hogy a C 64-es igen terjedelmes ROM-ja alá is tölthetünk programot. Töltés közben hol kikapcsolhatjuk, hol bekapcsolhatjuk (időnként kiiktathatjuk) a ROM-ot.

3.1.8 Autostart a megszakításokon keresztül

A C 64-es minden 1/60 másodpercben megszakítja az éppen futó programot, és végrehajt egy rutint. (A rutin feladata nagyon sokrétű; többek között a kurzor mozgását és villogását is ez intézi.) Ez a megszakítás IRQ (Interrupt ReQuest = megszakításkérés).

A processzor IRQ lábán minden 1/60 másodpercben érkező jel egy megszakítást vált ki. A processzor befejezi az éppen végrehajtás alatt álló utasítást, azután pedig a már ismertetett NMI megszakításhoz hasonlóan jár el. A különbség annyi, hogy mielőtt engedne a megszakításkérelemnek, megvizsgálja az állapotregiszter harmadik bitjének, az interrupt (I) kapcsolónak az értékét. Ha ez 1, visszautasítja a megszakításkérelmet. Ha a kapcsoló értéke 0 volt, akkor a processzor az előbb leírtak szerint jár el: az I kapcsolót 1-re állítja annak érdekében, hogy egy újabb megszakítás meg ne zavarhassa a „kitérő" programot. Betölti a $FFFE és $FFFF címek tartalmát (felső, alsó byte), és ezt a címet tekinti a programszámláló új értékének.

Az előző alfejezet tapasztalatai alapján ezt a lehetőséget is fel tudjuk használni az autostart-programozásra. Az előzőekhez hasonlóan módosítjuk a $0314/$0315 (788/799) címen található IRQ vektort is. Miután a megszakítás megtörtént és a vezérlés már a kezünkben van, állítsuk az I bitet 1-re a SEI utasítással. A továbbiakban megvizsgáljuk, hol tart a betöltés, és ha már ott, ahol nekünk kell, az előző alfejezet szerint járjunk el. Nem szabad a SEI utasítást kifelejteni, mert ekkor az újabb megszakításnál megint a program elejére ugrik a rendszer.

3.1.9 Autostart a CIA chipek felhasználásával

A címben szereplő állítás tulajdonképpen hamis: ez a módszer főleg nem az autostart-programozásban lesz felhasználható.

Még mielőtt rátérnénk a CIA chipek programozására, ismerkedjünk meg magukkal a chipekkel. A CIA 6526 (Complex Interface Adapter) a 65xx-es processzorcsalád egy I/O (INPUT/OUTPUT) eleme, amely egy soros 8 bites regisztert, két *kaszkádolható* 16 bites számlálót (timer), egy valósidejű órát és egy *diverse* vezérlővonalat tartalmaz. A CIA további 16 regiszterét a processzor átmeneti tárolóként használja. A C 64-esben két ilyen I/O elem található, az egyik a $DC0F címig, a másik pedig a $DD00-tól a $DD0F-ig terjedő tárterületet foglalja el. A következőkben ismertetjük a CIA 16 regiszterét:

A vezérlőregiszterek leírása *5. táblázat*

0. regiszter	A port Hozzáférés: READ/WRITE A regiszter tartalma az A (I/O) port állapotát tükrözi.
1. regiszter	B port Hozzáférés: READ/WRITE A regiszter tartalma a B (I/O) port állapotát tükrözi.
2. regiszter	A adatirány-regiszter Hozzáférés: READ/WRITE Ezzel a regiszterrel az A port mind a 8 vonalát bevitelre, ill. kihozatalra kapcsolhatjuk. A kapcsoláshoz a regiszter megfelelő bitjét 0-ra (bevitel) vagy 1-re (kihozatal) kell állítani.
3. regiszter	B adatirány-regiszter Hozzáférés: READ/WRITE A regiszter feladata azonos a 2. regiszter feladatával a B portra vonatkoztatva.
4. regiszter	A számáló alsó byte Hozzáférés: READ Olvasásnál a regiszter tartalma az A számláló pillanatnyi értékének alsó byte-ját adja vissza Hozzáférés: WRITE Írásnál megadhatjuk annak az értéknek az alsó byte-ját, amelyről a számláló nullára visszaszámlál.

5. regiszter

A számláló felső byte
Hozzáférés: READ
Visszakapjuk az A számláló pillanatnyi értékének alsó byte-ját.
Hozzáférés: WRITE
Író utasítással megadhatjuk annak az értéknek a felső byte-ját, amelyről az A számláló nullára visszaszámlál.

6. regiszter

B számláló alsó byte
Ua. mint a 4. regiszter a B számlálóra vonatkoztatva.

7. regiszter

B számláló felső byte
Ua. mint az 5. regiszter a B számlálóra vonatkoztatva.

8. regiszter

A valós idejű óra (time of day) tizedmásodpercei.
Hozzáférés: READ
Olvasásnál a 0-tól 3-ig terjedő bitek tartalma visszaadja a valós idejű óra pillanatnyi értékét tizedmásodpercekben, BCD kódban. A 4-től 7-ig terjedő bitek értéke mindig nulla.
Hozzáférés: WRITE
Írásnál a B vezérlőregiszter (15.) előzetes beállításával választhatunk: megadhatjuk az óra aktuális értékét tizedmásodpercekben, vagy a riasztás időpontját.
A 4-től 7-ig terjedő biteket nullára kell állítani, a tizedmásodperceket pedig BCD formátumban kell megadni.

9. regiszter

A valós idejű óra másodpercei
Hozzáférés: READ
Olvasásnál a regiszter tartalma visszàadja az óra pillanatnyi értékét másodpercekben, a 0-tól 3-ig terjedő bitek értéke az egyeneseket, a 4-től 7-ig terjedő bitek értéke pedig a tízeseket jelenti.
Hozzáférés: WRITE
Ua. mint a 8. regiszternél. A másodpercek formátuma is BCD kód.

10. regiszter

A valós idejű óra percei
Ua. mint a 9. regiszternél a percekre vonatkoztatva.

11. regiszter

A valós idejű óra órái
Hozzáférés: READ
Olvasásnál visszakapjuk az idő aktuális értékét órában. A 0-tól 3-ig terjedő bitek értéke most is az egyeseket

jelenti. Mivel az óra legnagyobb értéke 12 lehet, a tízesek jelölésére elég egyetlen bit, nevezetesen a 4. bit.
7. bit az amerikai időnek megfelelően a délelőtt (AM, 7. bit = 0), ill. délután (PM, 7. bit = 1) jelölésére szolgál.
Hozzáférés: WRITE
Ua. mint a fenti regisztereknél, de a biteket az olvasási hozzáférésnél leírtak szerint kell kezelni.

12. regiszter

Soros eltolási regiszter
Kiírásnál a soros buszra kerülő, olvasásnál a soros buszról érkező adatokat tárolja bitenként.

13. regiszter

Megszakítást vezérlő regiszter
(Interrupt Control Register)
Hozzáférés: READ
0. bit: az A számláló lefutása
1. bit: a B számláló lefutása
2. bit: az aktuális idő és a riasztási idő értéke azonos
3. bit: az eltolási regiszter megtelt (beolvasásnál) vagy üres (kiírásnál)
4. bit: a FLAG lábon impulzus érkezett
5–6. bit: mindig nulla
7. bit: értéke 1, ha a megszakítást vezérlő és a megszakítást maszkoló regiszterek 0-tól 4-ig terjedő bitjei közül legalább egy értéke 1.

Figyelem! Olvasásnál a regiszter tartalma törlődik!
Hozzáférés: WRITE (Megszakítási maszk)
A 0-tól 4-ig terjedő bitek jelentése azonos. Ha ráadásul a 7. bit értéke 1, a megszakítást tetszőleges művelet számára kihasználhatóvá tehetjük. Ha a 7. bit nulla, a megszakítás nem engedélyezett.

14. regiszter

A vezérlőregiszter
Hozzáférés: READ/WRITE
0. bit: 0 = A számláló STOP = 1 A számláló START
1. bit: 1 = az A számláló lefutását jelzi a PB6
2. bit: 0 = az A számláló minden lefutása előállít egy magas impulzust a PB6 lábon
1 = az A számláló minden lefutása megfordítja a PB6 állapotát

3. bit: 1 = az A számláló visszaszámlál a kezdőértéktől nulláig, majd leáll (one shot)
0 = az A számláló minden lefutás után automatikusan újraindul
4. bit: 1 = az A számláló új értékének betöltése feltétel nélkül
5. bit: 0 = az A számláló számlálja a rendszerütemet, 1 = számlálja a CNT ütemet
6. bit: 0 = a soros port bemenet, 1 = a soros port kimenet
7. bit: 0 = a valós óra 60 Hz-cel üzemel, 1 = a valós óra 50 Hz-cel üzemel

15. regiszter

B vezérlőregiszter
Hozzáférés: READ/WRITE
0–4. bitek: az A vezérlőregiszter bitjeivel azonos jelentésűek a B számlálóra és a PB7-re vonatkoztatva
5–6. bitek: ezek a bitek határozzák meg a B számláló triggerforrását:
00 = a számláló a rendszerütemet,
01 = a számláló a CNT ütemet számlálja
10 = a B számláló az A számláló lefutásait számlálja
11 = a B számláló az A lefutásait számlálja, ha CNT = 1
7. bit: 0 = valós idő beállítása
1 = riasztási idő beállítása.

Az eljárás lényege az, hogy a CIA chip segítségével kiváltunk egy NMI megszakítást. A megszakítás akkor következik be, ha a CIA egyik timere lefut, azaz nulla értéket vesz fel. Amikor a megszakítás kiváltódik, a rendszer a már említett módon használja az NMI vektorokat. Az NMI vektor sajnos nagyon messze van a CIA regisztereitől, ezért ennek a módszernek inkább a programvédelemben van igazi jelentősége, nem pedig az autostartban.

Hogyan is kell tulajdonképpen használni a CIA chipet? A B timer értékét a $DD06-os és $DD07-es tartalmazzák, a számláló ettől az értéktől kezdve számol egyesével lefelé. Amikor eléri a nullát, a $DD0F címből kivon egyet, és azonnal egy NMI megszakítást vált ki, azaz a $0318/$0319-es címen lévő vektor értékének megfelelő helyre ugrik. (A következő 5 regiszter most nem érdekel bennünket.)

A $DD0D címen a megszakításvezérlő (Interrupt Control) regiszter van. Ennek 1-es bitje jelzi a B számláló lefutását, ezért ezt a bitet állítsuk 1-re.

Ugyanennek a regiszternek a 7. bitje azt jelöli, hogy van-e 1 értékű bit a regiszteren belül, vagyis ezt is (magas) 1 állapotba kell hozni.

A következő regiszter szintén lényegtelen számunkra. A 15. regiszterről már szóltunk; e regiszter tartalma nagyon fontos. Nulladik bitjét 1-re kell állítani, ez jelzi a számláló indítását, az 5. és 6. biteket pedig 1-re. Az utóbbi két bit azt jelzi, hogy a számláló a rendszerütem ritmusa szerint csökken.

A $D006-tól $DD0F-ig terjedő címek tartalma tehát a következő:

DD00 xx xx xx xx xx xx 01 00
DD08 00 00 00 91 00 82 80 19

Ezeket az adatokat a tár bármely részén kimenthetjük. A kimentett adatokat a lemezen egy diskmonitorral kell majd megfelelő helyre tenni. Az adatok mögött kimenthetünk egy programot is. A program betöltés után a $3000 címen lesz a memóriában. Ahhoz, hogy a program tényleg a $3000-res címre töltődjön, az NMI vektor által meghatározott helyre olyan programot kell írni, amely az adatok betöltése után elindul, és „elintézi", hogy az utánuk levő program a valóban $3000-es címtől induljon. Az „intézkedő" program a következő:

```
2000 SEI         ;a megszakítások letiltása
2001 LDX #$00    ;az új célcím alsó byte-ja
2003 LDY #$30    ;             felső byte-ja
2005 STX $AE     ;alsó byte tárolása
2007 STY $AF     ;felső byte
2009 CLI         ;a megszakítás engedélyezése
200A RTI         ;visszaugrás (NMI)
```

3.1.10 Autostart címeltolással

Az előző fejezetekben számos példát láttunk arra, hogy a $AE/$AF (174/175) címek segítségével egy programot az eredeti betöltési címétől eltérő helyre töltsünk. A kívánt hatás eléréséhez a betöltendő byte címét alsó, felső byte alakban a $AE/$AF címekre kell tenni. Az iménti címek tartalmát nem kötelező programokból módosítani, megtehetjük ezt parancsmódból is, a program első néhány byte-jának betöltése után.

Az iménti két byte-nál tanácsos az alsó byte-ot változatlanul hagyni, és csak a felső byte-ot módosítani. Ha az alsó byte-ot módosítanánk és a felső byte értéke mindig $00 maradna, a betöltés a *nullláslapra* korlátozódna. Mivel a betöltést automatikus programindítással szeretnénk folytatni, további byte-okat kell elhelyezni a $0100 és $033C közötti tárterületen, pedig $0800-tól $FFFF-ig terjedő területet egy programrész már elfoglalt.

A megoldáshoz ismét némi trükkre van szükség. A $AE/$AF címeket módosítsuk úgy, hogy a $FE00+ a kívánt betöltési címre mutassanak. Az imént

említett területet pillanatnyilag ROM foglalja el, de ez nem okoz gondot, mivel az írás az ugyanazon a tárcímen elhelyezkedő, a ROM alatt fedésben levő RAM-ra történik. Ahhoz, hogy az ide írt programot elindíthassuk, ki kell kapcsolni a felette levő ROM-ot. A ROM kikapcsolását a következő utasításokkal érhetjük el:

```
LDA #$35
STA $01
```

A program betöltését e parancsok kiadása után egészen a $FFFF tárcímig folytathatjuk. Ha a program nem fér el ezen a területen, a $AE/$AF mutatót a célnak megfelelő értékre módosíthatjuk. Ha a töltés túlhalad a $FFFF címen, akkor a program töltését a $0000 címen folytatja. Nem blokkol le a rendszer abban az esetben sem, ha a nulláslapra azokat az adatokat írjuk vissza, amelyek eredetileg is ott voltak (a $AE/$AF címekre külön figyeljünk).

Az imént megismert technika befejeztével tetszőlegesen programozott autostartot is végrehajthatunk. A ROM-ot át kell kapcsolni RAM-ra, hogy hozzáférjünk. Az eljárást az Olvasó további hasznos ötletekkel kiegészítheti, hogy a program lényegét még jobban elrejtse az avatatlanok elől.

3.2 Az ellenőrző összeg és önmegsemmisítés

Ebben a részben az autostarttól teljesen eltérő módszereket fogunk ismertetni, amelyek abban az esetben próbálnak védelmet nyújtani, ha valaki már hozzáfért a programunkhoz. Az első eljárás úgy védi meg a programot, hogy felismeri az illetéktelen behatolást, és megakadályozza az átírt program működését. A behatolás felismerése igen egyszerű elven alapul. Képezzünk egy (a programra jellemző) számot úgy, hogy a program összes byte-ját összeadjuk; az így kapott számot nevezzük ellenőrző összegnek. Az ellenőrző összeget a program futása során többször kiszámolhatjuk, és helyességét megvizsgálhatjuk. Amennyiben úgy találjuk, hogy az összeg nem egyezik, különböző büntetéseket alkalmazhatunk a programba beavatkozók ellen.

Nézzünk egy példát az ellenőrző összeg használatára!

```
2000 LDY #$00      ;1. számláló beállítása
2002 LDX #$10      ;2. számláló beállítása
2004 LDA #$00      ;a kezdőcím alsó byte-jának betöltése
2006 STA $26       ;és tárolása
2008 LDA #$10      ;a kezdőcím felső byte-jának betöltése
200A STA $27       ;és tárolása
```

```
200C LDA ($26),Y ;érték betöltése
200E CLC         ;és hozzáadása
200F ADC $02     ;02 tartalmához
2011 STA $02     ;az eredmény tárolása 02-n
2013 INY         ;számláló növelése
2014 BNE $200C   ;ha nem 255 byte, köv. byte
2016 INC $27     ;felső byte növelése
2018 DEX         ;számláló csökkentése
2019 BNE $200C   ;ha a számláló nulla, tovább
101B LDA $02     ;a számított összeg betöltése
201D CMP #$36    ;és összehasonlítása az adott összeggel
201F BEQ $2029   ;ha egyenlők, vissza
2021 LDX #$00    ;számláló nullázása
2023 INC $D020   ;keretszín növelese
2026 INX         ;számláló növelése
2027 BNE $2023   ;ha 255-ször növeltük már, vissza
2029 RTS         ;visszaugrás
```

Ez a program a $1000–$1FFF tartalma alapján képezi az ellenőrző összeget. A $2002, $2004, $2008 sorokat változtatva, tetszés szerint változtathatjuk azt a területet, amelyik az ellenőrző összeg alapjául szolgál. A $201D sorban változtathatjuk meg az ellenőrző összeg értékét.

A program az adatok felvétele után ciklusban betölt egy adatot arról a címről, amire a $26/$27-es mutató mutat, és ezeket hozzáadja a $02-es tárcím tartalmához. Ha a $02-es tárcím értéke túlnő az egy byte maximális $FF-es értékén, akkor túlcsordulás keletkezik, azaz a byte értéke nulla lesz. A ciklus magját először addig ismételjük, amíg az Y regiszter értéke ismét nulla nem lesz. Abban az esetben, ha az Y értéke már nulla, a program az X értékét csökkenti eggyel, majd visszaugrik a ciklus elejére és ismét az Y-t csökkenti. Ha már az X regiszter tartalma is nulla, akkor a program végleg kilép a ciklusból és a $02-es byte értékét összehasonlítja a $201D sorban megadott ellenőrző összeggel. Ha az ellenőrző összeg nem hibás, a program visszaugrik a hívási cím mögé. Ha viszont hibát talált, akkor 255-ször növeli meg a keretszín regiszter tartalmát. Természetesen a villogtatás helyett egy sokkal nagyobb védelmet nyújtó rutinra cserélhetjük ki ezt a programrészt. Az egyik ilyen „megtorló rutin" lehet az, ami a programot törli ki a tárból vagy a lemezről.

```
033C SEI         ;megszakítás letiltása
033C LDA #$30    ;I/O terület és operációs rendszer
033F STA $01     ;kirekesztése
0341 LDX #$FB    ;1. számláló
0343 LDY #$00    ;2. szamlaló beállítása
0345 LDA #$00    ;a kezdőcím alsó byte-jának betöltése
0347 STA $26     ;és tárolása
0349 LDA #$04    ;a kezdőcím felső byte-jának betöltérse
034B STA $27     ;és tárolása
034D LDA #$00    ;érték betöltése
034F STA ($26),Y ;és tárolása
0351 INY         ;számláló növelése
0352 BNE $034F   ;ha 256 byte,
0354 INC $27     ;a kezdőcím felső byte-jának növelese
0356 DEX         ;szamlaló csökkentése
```

```
0357 BNE $034F     ;ha nulla,
0359 LDA #$37      ;operációs rendszer
035B STA $01       ;visszaengedése
035D CLI           ;megszakítás letiltása
035E JMP $FCE2     ;RESET-re ugrás
```

Az előbbi rutin a $0400-tól a $FFFF-ig terjedő területet nullákkal tölti fel, majd egy RESET rutint is meghív, és saját magát is kitörli. A program csak úgy tudja végrehajtani feladatát, ha az I/O területeket kikapcsolja, mivel azok írása leblokkolná a rendszert. Visszatérve az ellenőrző összegre, annak kiszámítását a közölt programhoz hasonlóval kell elvégezni, azzal a különbséggel, hogy nem kell bele az ellenőrző rész.

3.3 Kódolás programból

A monitorprogramok sok kódhoz nem rendelnek megfelelő assembler utasítást, ezek ún. nem megengedett műveleti kódok, részletesebben a 3.4 alfejezetben fogunk velük foglalkozni. Most pedig olyan eljárásokkal, amelyek hasonló elveken alapulnak és kitűnő lehetőséget nyújtanak a másolásvédelmek felismerésének megakadályozására. A következőkben a programból vezérelt kódolással ismerkedünk meg.

3.3.1 Az EXOR logikai művelet alkalmazása

Mielőtt a programkódolási eljárásokról beszélnénk, ismerkedjünk meg a programkódolás fogalmával. Programkódoláson egy program adatainak meghatározott rendszer szerinti átírását, „titkosítását" értjük. A kódolt programok a továbbiakban nem olvashatók a normál programokhoz hasonlóan, hacsak meg nem „fejtjük" a titkosítást, azaz nem dekódoljuk a programot. A titkosíráshoz hasonlóan a program titkosításához is egy olyan rendszer kell, amellyel a program byte-jait egy logikai művelet szerint átalakíthatjuk, és egy másik (vagy esetleg ugyanazzal) művelettel az eredeti állapotába hozhatjuk.

A kódolt programokat ahhoz, hogy futtatni tudjuk, a dekódoló programmal együtt kell tárolni. Természetesen a dekódoló rutint a program indítása előtt le kell futtatni, hogy visszaállítsa a byte-ok eredeti értékét. Ebben a fejezetben csak néhány kódolási módszert mutatunk be. A kódolási módszerek száma szinte végtelen. Minden programozó kitalálhat egy saját, csak az ő programjaiban alkalmazott titkosítási rendszert. Mi most a legegyszerűbb eljárásokat fogjuk bemutatni.

Az egyik legegyszerűbb módszer az EXOR-ral (EXCLUSIV OR – kizáró vagy) való kódolás. Az EXOR logikai művelet, a BOOLE-algebra egy művelete. A C 64-es gépi kódjában az EXOR műveletnek megfelelő utasítás az EOR.

Az EXOR műveletet a két byte (amelyekkel a műveletet végezzük) minden bitjével külön el kell végezni, és az eredménybitekből áll össze az eredménybyte. Az eredménybit minden olyan esetben 1, ha a műveletben részt vevő két bit eltérő. Nulla értéket akkor vesz fel az eredménybit, ha a két bit értéke megegyező.

0–0 = 0
0–1 = 1
1–0 = 1
1–1 = 0

Nézzük meg az EXOR művelet eredményét egy konkrét példán:

```
        A byte:    00110101 = $35 = 53
EXOR    B byte:    10101010 = $AA = 170
        eredmény:  10011111 = $9F = 159
```

Az EXOR művelet lényeges tulajdonsága, hogy ha elvégeztük az A és B byte-ok között, majd a műveletet megismételjük az eredmény és a B között, akkor visszakapjuk az A-t.

```
        A byte:    00110101 = $35 = 53
EXOR    B byte:    10101010 = $AA = 170
        eredmény:  10011111 = $9F = 159
EXOR    B byte:    10101010 = $AA = 170
        eredmény:  00110101 = $35 =  53
```

Az eredeti számot tehát csak a kódszám ismeretében lehet a kódolt számból megkapni. Az előző tulajdonság az oka, hogy a kódoló és a dekódoló rutin is ugyanaz. A következő program egy példa a kódoló és a dekódoló rutinra:

```
1000 LDX #$00       ;mutató nullázása
1002 LDA $2000,X    ;byte betöltése és
1005 EOR #$F4       ;(de)kódolása
1007 STA $2000,X    ;byte visszaírása
100A INX            ;mutató növelése
100B BNE $1002      ;ha < 256, a ciklus elejére
100D RTS            ;visszaugrás
```

A kódolni kívánt programot a $2000–30FF-ig terjedő részre tegyük. A kódoló rutin a dekódolást is elvégzi, ha a $100D címen levő RTS-t JMP-re cseréljük. Ha a kódolni kívánt program hosszabb 255 byte-nál, akkor az egészet tegyük be egy új ciklusba.

```
2027 LDX #$00      ;1. mutató beállitasa
2039 LDY #$00      ;2. mutató beállítasa
202B LDA $5000,X ;programbyte betöltése
202E EOR #$F4      ;a művelet
2030 STA $5000,X ;a byte újboli tárolása
2033 INX           ;1. mutató növelese
2034 BNE $202B     ;cikluskezdetre ha < 256
2036 INC $202D     ;prog. száml. felső byte-jának növelese
2039 INC $2032     ;célcím felső byte-jának növelese
203C INY           ;2. mutató növelése
203D CPY #$10      ;$10 blokk (de)kódolva?
203F BNE $202B     ;cikluskezdetre, ha nem
2041 RTS           ;visszaugrás
```

3.3.2 Kódolás rejtett beugrással

Az előző alfejezetben leírt módszer nagy előnye az egyszerűség, a kevésbé rutinos programozók is könnyen tudják hasznosítani programjaikban. Az egyszerűség ára, hogy az ezzel a módszerrel védett program tulajdonképpen elég gyengén védett. Gondoljuk csak meg, mi történne akkor, ha valaki a dekódoló rutin végén a programra ugrás helyett a saját rutinjára ugrik, vagy megállítja a programot egy BRK utasítással. A dekódoló rutin a módosítás miatt leáll, de a feladatát előbb elvégzi. A program ott áll teljesen szabályosan. Ebben az esetben a programunk teljesen védtelenül hozzáférhetővé válik bárki számára.

A problémák megoldása csak egy lehet: az ugró utasítást valahogy el kell rejteni a beavatkozó elől. A következő program megkísérli az ugró utasítást elrejteni:

```
1000 LDX #$FF     ;a mutató FF-re  állítása
1002 LDA 1000,X ;kódolt byte betöltése
1005 EOR #$F4     ;dekódolása F$-el
1007 STA 1000,X ;byte visszaírása
100A DEX          ;mutató csökkentése 1-gyel
100B BNE $1002    ;a ciklus kezdetre
100D BRK          ;látszólagos programvég
```

Reméljük, senki sem találta ki, hogy a dekódolás végeztével hová is ugrik a rutin, azaz hol kezdődik a főprogram. A dekódoló rutin az eddigiektől eltérően hátulról kezdi a tárterület dekódolását. A dekódolás akkor ér véget, ha az X regiszter értéke nulla lesz. Mielőtt az X regiszter értéke nulla lehetne, a rutin saját magát is módosítja. Ebben áll a módszer lényege. Amikor a BNE utasítást próbálja a rutin megváltoztatni, akkor csak azt a részét írja át, ami az ugrási címet tartalmazza, mert ez van a magasabb tárcímen. A BNE egy két byte-os gépi kódú utasítás. Első byte-ja magát az utasítást és a címzésmódot jelöli. A második byte a relatív ugrási címet. A második byte 128-nál kisebb értéke azt jelenti, hogy ennyit kell ugrani előre. Ha a byte 128-nál nagyobb, akkor le kell vonni 128-at a byte-ból; a kapott szám a hátraugrás mértékét jelöli. Össze-

foglalva, az ugrás mértékét az alsó 7 bit értéke adja, a legfelső bit pedig az ugrás irányát jelöli. A $100B címen levő BNE mindaddig a ciklus visszatérési utasítása (a $1002-es címre ugrik vissza), amíg a dekódoló rutin meg nem változtatja az ugrási címét. A BNE megváltoztatott tárcímének a már dekódolt program elejére kell mutatnia. A mi esetünkben a program a $100E címen kezdődik, a BNE új ugrási címének tehát ide kell mutatnia. Az elmondottak alapján már nyilvánvaló, hogy a BNE második byte-ja és a $F4 között kell elvégezni az EXOR műveletet, hogy az ugrás a $100E címre történjen. A programlistán látható, hogy valóban a $F4-et használjuk fel a dekódoláshoz.

A titkosítás jobb hatású, ha valamivel bonyolultabb rutinnal próbáljuk „megkeverni" azt, aki fel akarja törni programunkat.

A módszer kipróbálására írjuk a $100E-től kezdődően a következő programot:

```
100E LDX #$00     ;1. mutató
1010 LDY #$00     ;2. mutató beállítása
1012 INC $D020    ;keretszín növelese
1015 INX          ;1. mutató csökkentése
1016 BNE $1012    ;cikluskezdetre
1018 INY          ;2. számlaló növelése
1019 BNE $1012    ;cikluskezdetre
101B BRK          ;programvég
```

Kódoljuk ezt a programot a következő rutin segítségével:

```
2000 LDX #$0E      ;mutató 0E-re állítása
2002 LDA $1000,X   ;első byte be
2005 EOR #$F4      ;a byte kódolása
1007 STA $1000,X   ;a byte visszaírása
200A INX           ;mutató növelese
200B BNE $2002     ;a cikluskezdetre
200D RTS           ;visszaugrás
```

Természetesen a kódolás a $100E címtől kezdődik, mert előtte a dekódoló rutin van.

A kódolt program és a dekódoló rutin így néz ki a tárban:

```
1000 LDX #$FF      ;mutató beállítasa
1002 LDA $1000,X   ;byte betöltése
1005 EOR #$F4      ;és. dekódolása
1007 STA $1000,X   ;byte visszaírása
100A DEX           ;mutató csökkentése
100B BNE $100E,X   ;cikluskezdetre
100D ???           ;látszólagos programvég
100E LSR $F4,X     ;kódolt program
1010 ???           ;
1011 ???           ;
1012 ???           ;
1013 ???           ;
1014 BIT $1C       ;
1016 BIT $0E       ;
1018 ???           ;
1019 BIT $03       ;
```

```
101B ???          ;
101C ASL $1E1E,X  ;
101F ASL $1E1E,X  ;a kódolt program vége
```

3.3.3 Kódolás a timerrel

Az előző fejezetekben egyszerű kódoló, dekódoló programokról volt szó. A továbbiakban egy sokkal bonyolultabb rendszerrel fogunk megismerkedni. Eljárásunk a már ismertetett CIA chipet fogja felhasználni. Azt szeretnénk kivédeni az eljárással, hogy a kód megfejtése után bárki módosíthassa a programot. Nem állandó kóddal fogunk dolgozni, így a kódolt program visszafejtése is sokkal nehezebb lesz. Az eljárást a CIA-k ütemciklusa alapján fogjuk végezni. Az első felmerülő probléma, hogy a kódoló és dekódoló rutinok időben nem térhetnek el egymástól. Az iménti nehézségeket a következő programmal illusztráljuk:

```
2000 SEI          ;megszakitas letiltása
2001 LDA #$00     ;érték betöltése
2003 STA $DD0F    ;és a timer megállítása
2006 LDA #$FF     ;az alsó byte
2008 STA $DD06    ;timerbe írása
200B LDA #$FF     ;a felsó byte
200D STA $DD07    ;timerbe írása
2010 LDA #$99     ;érték írása a
2012 STA $DD0F    ;timer continue-mód indító regiszterbe
2015 LDX #$00     ;mutató nullázása
2017 INX          ;mutató növelese
2018 BNE $2017    ;cikluskezdetre, ha < 256
201A LDA $DD06    ;timer alsó byte betöltése és
201D STA $4000    ;tárolása 4000-en
2020 LDA $DD07    ;felsó byte betöltése és
2023 STA $4001    ;tarolása 4001-en
2026 LDA #$00     ;érték betöltése és
2028 STA $DD0F    ;timer megállitasa
202B CLI          ;megszakItas engedélyezése
202C RTS          ;visszaugrás
```

A program meghatározott ideig futtatja a B számlálót, ezután az értékét a $4000, $4001-es címeken tárolja. A következő BASIC program elindítja a gépi kódú rutint, és a $4000, C$4001-es tárcímek értékét (ahol a számlálók értékeit tároltuk) a képernyőre írja.

```
10 SYS2*4096
20 PRINTPEEK(4*4096),PEEK(4*4096+1)
30 GOTO10

READY.
```

A képernyőre írt érték állandóan változik, holott a számnak állandónak kellene lennie, hiszen ugyanarról az értékről indítjuk és a program sem változik közben. A változás oka, hogy a programmal ellentétben a timer folyamatosan működik. A program elején egy SEI utasítással letiltottuk a program megszakítását, de a videovezérlő azért időnként kénytelen megszakítani a programunkat. A videovezérlő feladata a televíziós kép létrehozása. Kapcsoljuk hát ki a képernyőt, így megakadályozva a további megszakításokat. A képernyő kikapcsolását a VIC chip 17. regiszterében levő 4. bit nullázásával érhetjük el. A 17. regiszter a $D011-es címen van.

A képernyő kikapcsolását csak a raszterletapogatás lefutása után végezhetjük el, ezért a következő programba a $2009-től $2011-ig terjedő sorokba egy várakozó ciklust írtunk.

```
2000 SEI          ;megszakítas letiltása
2001 LDA $D011    ;érték az 1. vezérlő regiszterből
2004 AND #$EF     ;a 4. bit törlése, a képernyő kikapcsolasa
2006 STA $D011    ;
2009 LDX #$00     ;a váró ciklus 1. mutatója
200B LDY #$F0     ;               2. mutatója
200D INX          ;1. mutató növelese
200E BNE $200D    ;ha < 256, cikluskezdete
2010 INY          ;2. mutató növelese
2011 BNE $200D    ;ha < 256, cikluskezdete
2013 LDA #$00     ;érték betöltése és
2015 STA $DD0F    ;timer megállítása
2018 LDA #$FF     ;timer alsó byte-ja
201A STA $DD06    ;beállítása
201D LDA #$FF     ;timer felső byte-ja
201F STA $DD07    ;beállítása
2022 LDA #$99     ;timer continue-mód
2024 STA $DD0F    ;elindítása
2027 LDX #$00     ;mutató beállítása
2029 INX          ;mutató növelése
202A BNE $2029    ;ha < 256, cikluskezdete
202C LDA #$00     ;timer
202E STA $DD0F    ;megállítása
2031 LDA $D011    ;képernyő
2034 ORA #$10     ;vissza-
2036 STA $D011    ;kapcsolása
2039 LDA $DD06    ;timer alsó byte a
203C STA $4000    ;4000-re
203F LDA $DD07    ;felső byte a
2042 STA $4001    ;4001-re
2045 CLI          ;megszakítás engedélyezése
2046 RTS          ;visszaugrás
```

Ezt a rutint is ugyanazzal a BASIC rutinnal indíthatjuk, amellyel az előzőt. Az indítás után a timerből leolvasott érték állandó lesz.

Miután a timer működését már jól tudjuk irányítani, térjünk vissza a kódoláshoz. A kódolást az előzőekhez hasonlóan most is az EXOR műveletekkel fogjuk elvégezni, azzal a különbséggel, hogy amíg eddig állandó értékkel dolgoztunk, most a timer éppen aktuális értékével fogunk. Az állandóan változó timerrel

végzett művelet azt eredményezi, hogy egy azonos adatokból létrehozott új adatsorban szereplő értékek különbözőek lesznek. Az eljárás csak úgy működhet, ha a timer a kódoló és a dekódoló rutin során is ugyanarról az értékről indul. A kódoló rutin a következő:

```
2000 SEI             ;megszakítas letiltása
2001 LDA $D011       ;képernyő
2004 AND #$EF        ;kikapcsolása
2006 STA $D011       ;(a 4-ik bit törlése)
2009 LDX #$00        ;1. mutató
200B LDY #$F0        ;2. mutató betöltése
200D INX             ;1. mutató növelese
200E BNE $200D       ;ha < 256, cikluskezdetre
2010 INY             ;2. mutató növelese
2011 BNE $200D       ;ha < 256, cikluskezdetre
2013 LDA #$00        ;timer
2015 STA $DD0F       ;megállítása
2018 LDA #$FF        ;timer alsó byte-ja
201A STA $DD06       ;
201D LDA #$FF        ;timer felsó byte-ja
201F STA $DD07       ;beállítása
2022 LDA #$99        ;timer continue-mód
2024 STA $DD0F       ;elindítása
2027 LDX #$00        ;1. mutató
2029 LDY #$F0        ;2. mutató beállítása
202B LDA $5000,X     ;program byte betöltése
202E EOR $DD06       ;EOR elvégzése a timer alsó byte-tal
2031 STA $5000,X     ;program byte tárolása
2034 INX             ;1. mutató növelese
2035 BNE $202B       ;ha < 256, ciklus kezdetre
2037 INC $202D       ;programszámláló felsó byte növelese
203A INC $2033       ;célcím felsó byte növelése
203D INY             ;2. mutató növelese
203E BNE $202B       ;ha < 256, ciklus kezdetre
2040 LDA #$50        ;a program és célcím
2042 STA $202D       ;eredeti értékének
2045 STA $2033       ;visszaállítása
2048 LDA #$00        ;timer
204A STA $DD0F       ;megállítása
204D LDA $D011       ;képernyó
2050 ORA #$10        ;vissza-
2052 STA $D011       ;kapcsolása
2055 CLI             ;megszakítás engedélyezése
2056 RTS             ;visszaugrás
```

Tegyük próbára a rutint. Legyen a kódolandó adatsor a következő:

```
5000 55 55 55 55 55 55 55 55
5008 55 55 55 55 55 55 55 55
5010 55 55 55 55 55 55 55 55
5018 55 55 55 55 55 55 55 55
5020 55 55 55 55 55 55 55 55
5028 55 55 55 55 55 55 55 55
5030 55 55 55 55 55 55 55 55
```

A futtatás után ugyanez a tárcím a következő lesz:

```
5000 A3 B1 87 95 FB C9 DF 2D
5008 33 01 17 65 4B 59 AF BD
5010 83 91 E7 F5 BD 29 3F 0D
5018 13 61 77 45 AB B9 8F 9D
5020 E3 F1 C7 D5 3B 09 1F 6D
5028 73 41 57 A5 8B 99 3F FD
5030 C3 D1 27 35 1B 69 7F 4D
```

A dekódolást ugyanezen elv alapján végezhetjük el.

3.3.4 Egylépéses dekódolás

A kódtörők egyik nagyon hatásos eszköze a védelmek ellen, hogy RESET gombbal látják el gépüket. Az ilyen módon „kívülről" megállított programok teljesen ki vannak szolgáltatva a programfejtőknek; védeni szánt programjainkat tehát erre az eshetőségre is fel kell készítenünk. Eljárásunk – amely egyebek között a RESET gombok ellen is megpróbál védelmet nyújtani – abban különbözik a már ismertektől, hogy soha sincs az egész program dekódolva, hanem csak az éppen futó utasítás. A rutin működése úgy foglalható össze, hogy az éppen végrehajtott utasítást újra kódolja, majd a következőt dekódolja. A programból tehát mindig csak egészen kis rész „érthető". Ha valakinek sikerülne is a programot megállítani, a kódolás miatt nem fogja megérteni.

A „titkosítani" kívánt program első sorának tartalmaznia kell a JSR $C000 utasítást, mert ez aktivizálja a dekódoló programot. Ezen kívül azt is be kell tartani, hogy a program első hat byte-ját nem szabad kódolni, mert akkor nem tudnánk elindítani a programot. A következő rutin egy tetszőleges program kódolását végzi:

```
C000 LDA #$74      ;1:C074 alsó byte
C002 LDY #$C0      ;1:C074 felső byte
C004 JSR $AB1E     ;szöveg kivitele
C007 JSR $C03E     ;egy szám bevitele
C00A CLC           ;carry törlése összeadása
C00B ADC #$06      ;06 hozzáadása
C00D STA $FB       ;tarolása a prog. mut. alsó byte-ban
C00F TXA           ;a felső byte bevitele
C012 STA $FC       ;tárolása a prog. mut. felső byte-ban
C014 LDA #$85      ;2:C085 alsó byte
C016 LDY #$C0      ;2:C085 felső byte
C018 JSR $AB1E     ;szöveg kivitele
C01B JSR $C03E     ;egy szám bevitele
C01E STA $FD       ;alsó byte tárolása
C020 STX $FE       ;felső byte tárolása
C022 LDY #$00      ;index nullázása
C024 LDA ($FB),Y   ;program byte betöltése
C026 JSR $C096     ;byte kódolása
C029 STA ($FB),Y   ;tárolása
```

```
C02B INC $FB      ;mutató alsó byte növelése
C02D BNE $C031    ;elágazás, ha nincs átvitel
C02F INC $FC      ;mutató felső byte növelése
C031 LDA $FB      ;mutató alsó byte betöltése
C033 CMP $FD      ;összehasonlítása a végcímmel
C035 BNE $C024    ;elágazás, ha nem egyenlő
C037 LDA $FC      ;mutató felső byte betöltése
C039 CMP $FE      ;összehasonlítása a végcímmel
C03B BNE $C024    ;elágazás, ha nem gyenlő
C03D RTS          ;visszaugrás
```

A következő rutinok a kódoló rutin alprogramjai.

Egy négyjegyű hexadecimális szám beolvasása:

```
C03E JSR $C042   ;kétjegyű szám bevitele
C041 TAX         ;és X-be tétele alsó byte-ként
C042 JSR $C052   ;egy szám bevitele
C045 ASL $A      ;negyszeres
C046 ASL $A      ;lánc-SHIFT-tel
C047 ASL $A      ;a fenti
C048 ASL $A      ;fél byte-ba
C049 STA $C050   ;és tárolása OR művelet számára
C04C JSR $C052   ;egy szám bevitele
C04F ORA #$00    ;összehasonlítása a fenti fél byte-tal
C051 RTS         ;visszaugrás
```

Egy hexadecimális számjegy beolvasása és átalakítása:

```
C052 JSR $FFCF   ;BASIC: adott karakterek betöltése
C055 SEC         ;carry beállítása kivonáshoz
C056 BSC #$30    ;$30 kivonása
C058 BCC $C069   ;elágazás, ha túl kicsi
C05A CMP #$0A    ;összahasonlítása 0A-val
C05C BCC $C068   ;elágazás, ha kisebb
C05E SBC #$07    ;$07 kivonása
C060 CMP #$0A    ;összehasonlítása 0A-val
C062 BCC $C069   ;elágazás, ha túl kicsi
C064 CMP #$10    ;összehasonlítása $10-zel
C066 BSC $C069   ;elágazás, ha túl nagy
C068 RTS         ;visszaugrás
C069 LDX #$F8    ;$F8 az X-be
C06B TXS         ;veremmutató inicializálása
C06C LDA #$3F    ;ASCII "?"
C06E JSR $FFD2   ;a kérdőjel kivitele
C071 JMP $A474   ;vissza a BASIC értelmezőhöz
C074 93 4B 45 5A 44 4F 43 49  .KEZDOCI
C07C 4D 20 20 20 20 3A 20 24  M    : $
C084 00 0D 56 45 47 43 49 4D  ..VEGCIM
C08C 20 20 20 20 2B 31 3A 20      +1:
C094 24 00 18 65 FB 49 55 60  $.....
```

A kódoló program kódolást végző alprogramja:

```
C096 CLC         ;carry törlése összeadáshoz
C097 ADC $FB     ;program számláló alsó byte hozzáadása
C099 EOR #$55    ;EXOR $55 művelet
C09B RTS         ;visszaugrás
```

A kódoló program BASIC betöltője:

```
10.FORI=0 TO164
20 READ A$:A$=MID$(A$,1,2):B$=RIGHT$(A$,1):C$=LEFT$(A$,1)
30 B=ASC(B$)-48:IFB>9THENB=B-7
40 C=ASC(C$)-48:IFC>9THENC=C-7
50 D=16*C+B:POKE28672+I,D
60 NEXT
100 DATA A9,74    :REM       LDA #$74
110 DATA A0,C0    :REM       LDY #$C0
120 DATA 20,1E,AB:REM        JSR $AB1E
130 DATA 20,3E,C0:REM        JSR L1
140 DATA 18       :REM       CLC
150 DATA 69,06    :REM       ADC #$06
160 DATA 85,FB    :REM       STA $FB
170 DATA 8A       :REM       TXA
180 DATA 69,00    :REM       ADC #$00
190 DATA 85,FC    :REM       STA $FC
200 DATA A9,85    :REM       LDA #$85
210 DATA A0,C0    :REM       LDY #$C0
220 DATA 20,1E,AB:REM        JSR $AB1E
230 DATA 20,3E,C0:REM        JSR L1
240 DATA 85,FD    :REM       STA $FD
250 DATA 86,FE    :REM       STX $FE
260 DATA A0,00    :REM       LDY #$00
270 DATA B1,FB    :REM C2    LDA ($FB),Y
280 DATA 20,96,C0:REM        JSR L2
290 DATA 91,FB    :REM       STA ($FB),Y
300 DATA E6,FB    :REM       INC $FB
310 DATA D0,02    :REM       BNE C1
320 DATA E6,FC    :REM       INC $FC
330 DATA A5,FB    :REM C1    LDA $FB
340 DATA C5,FD    :REM       CMP $FD
350 DATA D0,ED    :REM       BNE C2
360 DATA A5,FC    :REM       LDA $FC
370 DATA C5,FE    :REM       CMP $FE
380 DATA D0,E7    :REM       BNE C2
390 DATA 60       :REM       RTS
400 DATA 20,42,C0:REM L1     JSR L3
410 DATA AA       :REM       TAX
420 DATA 20,52,C0:REM L3     JSR L4
430 DATA 0A       :REM       ASL
440 DATA 0A       :REM       ASL
450 DATA 0A       :REM       ASL
460 DATA 0A       :REM       ASL
470 DATA 8D,50,C0:REM        STA $C050
480 DATA 20,52,C0:REM        JSR L4
490 DATA 09,00    :REM       ORA #$00
500 DATA 60       :REM       RTS
510 DATA 20,CF,FF:REM L4     JSR $FFCF
520 DATA 38       :REM       SEC
530 DATA E9,30    :REM       SBC #$30
540 DATA 90,0F    :REM       BCC C3
550 DATA C9,0A    :REM       CMP #$0A
560 DATA 90,0A    :REM       BCC C4
570 DATA E9,07    :REM       SBC #$07
580 DATA C9,0A    :REM       CMP #$0A
590 DATA 90,05    :REM       BCC C3
600 DATA C9,10    :REM       CMP #$10
610 DATA B0,01    :REM       BCS C3
```

```
620 DATA 60        :REM C4      RTS
630 DATA A2,F8     :REM C3      LDX #$F8
640 DATA 9A        :REM         TXS
650 DATA A9,3F     :REM         LDA #$3F
660 DATA 20,D2,FF:REM           JSR $FFD2
670 DATA 4C,74,A4:REM           JMP $A474
680 DATA 93,4B,45:REM·   .KE
690 DATA 5A,44,4F:REM    ZDO
700 DATA 43,49,4D:REM    CIM
710 DATA 20,20,20:REM
720 DATA 20,3A,20:REM     :
730 DATA 24,00,0D:REM    $..
740 DATA 56,45,47:REM    VEG
750 DATA 43,49,4D:REM    CIM
760 DATA 20,20,20:REM
770 DATA 20,2B,31:REM      +1
780 DATA 3A,20,24:REM     : $
790 DATA 00        :REM    .
880 DATA 18        :REM L2      CLC
890 DATA 65,FB     :REM         ADC $FB
900 DATA 49,55     :REM         EOR #$55
910 DATA 60        :REM         RTS
920 DATA 00        :REM         BRK
930 DATA 00        :REM         BRK
940 DATA 00        :REM         BRK
950 DATA 00        :REM         BRK
960 DATA 00        :REM         BRK
970 DATA 00        :REM         BRK
980 DATA 00        :REM         BRK
990 DATA 00        :REM         BRK
1000 DATA00        :REM         BRK

READY.
```

Az előző kódoló programhoz tartozó dekódoló rutin az A timert használja arra, hogy az éppen végrehajtott utasítás után megszakítást generáljon. A megszakításban a már végrehajtott utasítást kódolja, a most következő pedig dekódolja:

```
C000 SEI          ;megszakítás letiltása
C001 LDA #$37     ;C037 az
C003 STA $0314    ;IRQ vektorba
C006 LDA #$C0     ;
C008 STA $0315    ;
C00B PLA          ;visszaugrasi cím alsó byte a veremből
C00C TAX          ;ennek megőrzése
C00D CLC          ;carry törlése összeadáshoz
C00E ADC #$01     ;egy hozzáadása
C010 STA $AC      ;tárolása a prog. mut. alsó byte-ban
C012 PLA          ;visszaugr. cím felső byte a veremből
C013 TAY          ;megőrzése
C014 ADC #$00     ;átvitel hozzáadása
C016 STA $AD      ;és tárolása a prog. mutatóban
C018 TYA          ;visszaugr. cím felső byte
C019 PHA          ;visszatétele a verembe
C01A TXA          ;visszaugr. cím alsó byte
C01B PHA          ;visszatétele averembe
C01C LDA #$16     ;0016 alsó byte-jának tárolása
C01E STA $DC04    ;az IRQ-timer alsó byte-jába
C021 LDA #$00     ;felső byte
```

```
C023 STA $DC05 ;tárolása
C026 LDA #$19  ;$19 az A-ba
C028 STA $DC0E ;és a timer elindításatása
C02B LDA $DC0D ;megszakítás kontról reg. törlése
C02E LDX #$02  ;idő ciklus számláló beállítása
C030 DEX       ;számláló csökkentése
C031 BNE $C030 ;elágazás, ha nem nulla
C033 BIT $FF   ;három ciklus kivárása
C035 CLI       ;megszakítás engedélyezése
C036 RTS       ;visszaugrás
```

A program BASIC betöltője:

```
100 FORI=1TO132STEP15:FORJ=0TO14:READA$:B$=RIGHT$(A$,1)
105 A=ASC(A$)-48:IFA>9THENA=A-7
110 B=ASC(B$)-48:IFB>9THENB=B-7
120 A=A*16+B:C=(C+A)AND255:POKE28671+I+J,A
125 NEXT:READA:IFC=ATHENC=0:NEXT:END
130 PRINT"HIBAS SOR:";PEEK(63)+PEEK(64)*256:STOP
300 DATA 78         :REM        SEI
310 DATA A9,37      :REM        LDA #$37
320 DATA 8D,14,03   :REM        STA $0314
330 DATA A9,C0      :REM        LDA #$C0
340 DATA 8D,15,03   :REM        STA $0315
350 DATA 68         :REM        PLA
360 DATA AA         :REM        TAX
370 DATA 18         :REM        CLC
380 DATA 69,01      :REM        ADC #$01
390 DATA 85,AC      :REM        STA $AC
400 DATA 68         :REM        PLA
410 DATA A8         :REM        TAY
420 DATA 69,00      :REM        ADC #$00
430 DATA 85,AD      :REM        STA $AD
440 DATA 98         :REM        TYA
450 DATA 48         :REM        PHA
460 DATA 8A         :REM        TXA
470 DATA 48         :REM        PHA
480 DATA A9,16      :REM        LDA #$16
490 DATA 8D,04,DC   :REM        STA $DC04
500 DATA A9,00      :REM        LDA #$00
510 DATA 8D,05,DC   :REM        STA $DC05
520 DATA A9,19      :REM        LDA #$19
530 DATA 8D,0E,DC   :REM        STA $DC0E
540 DATA AD,0D,DC   :REM        LDA $DC0D
550 DATA A2,02      :REM        LDX #$02
560 DATA CA         :REM  C1    DEX
570 DATA D0,FD      :REM        BNE C1
580 DATA 24,FF      :REM        BIT $FF
590 DATA 58         :REM        CLI
600 DATA 60         :REM        RTS
610 DATA A0,00      :REM        LDY #$00
620 DATA B1,AC      :REM C2     LDA ($AC),Y
630 DATA 20,6C,C0   :REM        JSR L1
640 DATA 91,AC      :REM        STA ($AC),Y
650 DATA C8         :REM        INY
660 DATA C0,03      :REM        CPY #$03
670 DATA D0,F4      :REM        BNE C2
680 DATA BA         :REM        TSX
690 DATA BD,05,01   :REM        LDA $0105,X
700 DATA 85,AC      :REM        STA $AC
```

```
710 DATA BD,06,01  :REM          LDA $0106,X
720 DATA 85,AD     :REM          STA $AD
730 DATA A0,00     :REM          LDY #$00
740 DATA B1,AC     :REM C3       LDA ($AC),Y
750 DATA 20,78,C0  :REM          JSR L2
770 DATA 91,AC     :REM          STA ($AC),Y
780 DATA C8        :REM          INY
790 DATA C0,03     :REM          CPY #$03
800 DATA D0,F4     :REM          BNE C3
810 DATA A9,11     :REM          LDA #$11
820 DATA 8D,0E,DC  :REM          STA $DC0E
830 DATA AD,0D,DC  :REM          LDA $DC0D
840 DATA 68        :REM          PLA
850 DATA A8        :REM          TAY
860 DATA 68        :REM          PLA
870 DATA AA        :REM          TAX
880 DATA 68        :REM          PLA
890 DATA 40        :REM          RTI
900 DATA 18        :REM          CLC
910 DATA 65,AC     :REM L1       ADC $AC
920 DATA 8C,74,C0  :REM          STY $C074
930 DATA 18        :REM          CLC
940 DATA 69,FF     :REM          ADC #$FF
950 DATA 49,55     :REM          EOR #$55
960 DATA 60        :REM          RTS
970 DATA 49,55     :REM L2       EOR #$55
980 DATA 8C,7F,C0  :REM          STY $C07F
990 DATA 38        :REM          SEC
1000 DATA E9,FF    :REM          SBC #$FF
1010 DATA 38       :REM          SEC
1020 DATA E5,AC    :REM          SBC $AC
1030 DATA 60       :REM          RTS
1040 DATA 00       :REM          BRK
1050 DATA 00       :REM          BRK
1060 DATA 00       :REM          BRK

READY.
```

A következő rutinok a dekódoló alegységei.

A dekódoló megszakítási rutinjai:

```
C037 LDY #$00      ;index nullázása
C039 LDA ($AC),Y   ;programbyte bevitele
C03B JSR $C06C     ;byte kódolása
C03E STA ($AC),Y   ;és tárolása
C040 INY           ;programszámláló növelése
C041 CPY #$03      ;van-e 3 kódolt byte?
C043 BNE $C039     ;elágazás ha nincs
C045 TSX           ;veremmutató az X-be
C046 LDA $0105,X   ;programsxzámláló alsó byte betöltése
C049 STA $AC       ;és megőrzése
C04B LDA $0106,X   ;a felső byte
C04E STA $AD       ;
C050 LDY #$00      ;index nullázása
C052 LDA ($AC),Y   ;programbyte bevitele
C054 JSR $C07B     ;byte dekódolása
C057 STA ($AC),Y   ;és tárolása
C059 INY           ;programszámláló növelese
C05A CPY #$03      ;van-e 3 kódolt byte
C05C BNE $C052     ;elágazás, ha nincs
```

```
C05E LDA #$11      ;$11 az A-ba
C060 STA $DC0E     ;és a timer elindítása
C063 LDA $DC0D     ;megszakítás kontrolregiszter törlése
C066 PLA           ;Y reg. feltöltése
C067 TAY           ;a veremből
C068 PLA           ;X reg. feltöltése
C069 TAX           ;a veremből
C06A PLA           ;A a veremből
C06B RTI           ;vissatérés
```

A dekódolást végző alprogram:

```
C078 EOR #$55   ;EXOR #$55 művelet
C07A STY $C07F  ;X reg. tárolása a kivonáshoz
C07D SEC        ;carry beállítása kivonáshoz
C07E SBC #$FF   ;Y reg kivonása
C080 SEC        ;carry beállíytása kivonáshoz
C081 SBC $AC    ;programszámláló alsó byte kivonása
C083 RTS        ;visszaugrás
```

Az újrakódoló program:

```
C06C CLC        ;carry törlése összeadáshoz
C06D ADC $AC    ;programszámláló alsó byte hozzÁadása
C06F STY $C074  ;Y reg. tárolása összeadáshoz
C072 CLC        ;carry törlése összeadáshoz
C073 ADC #$FF   ;Y reg hozzáadása
C075 EOR #$55   ;EXOR #$55 művelet
C077 RTS        ;visszaugrás
```

A kódolandó program írásakor néhány kisebb korlátozást is figyelembe kell venni. Már említettük, hogy a programot JSR $C000 utasítással kell kezdeni. Ha olyan utasítást tartalmaz, ami saját magát módosítja, akkor külön figyelni kell arra, hogy kódolva vagy anélkül módosítjuk a byte-ok tartalmát. A programban szereplő adatokat sem szabad kódolni, mert ezeket a dekódoló nem alakítja vissza. Természetesen írható olyan program, amiben a dekódoló az adatokkal is foglalkozik. A billentyűzet lekérdezése a program folyamán le van tiltva; az operációs rendszerre hivatkozhatunk; a ROM-ot nem tudjuk megváltoztatni. Mind a kódoló, mind a dekódoló a ROM-ra való hivatkozás során az alatta levő RAM-ot írja át.

A dekódolást egy program futása során fel is függeszthetjük. Mivel a dekódolás megszakításból történik, a SEI utasítás megszünteti (letiltja) a megszakítást. A dekódolás megszüntetésével lehetőségünk van kódolatlan programrészek beépítésére. A kódolatlan programrészekre akkor van szükség, ha időzítési problémáink vannak, vagy ha ezzel nagyon megnövelnénk a futási időt. A dekódolást a CLI utasítással lehet folytatni.

Hogy a dekódolásból végleg kiléphessünk, és befejezhessük a programot, a következő három utasítást kell használni:

```
SEI
JSR $FF84
JSR $FF8A
```

Amikor a kódoló rutin a végcímet kéri, a $FF8A-t adjuk meg.

Néhány tudnivaló a használathoz: legelőször a kódoló program BASIC betöltőjét indítsuk el, ezután töltsük be a kódolandó programot. Töltsük a tárba a dekódoló program BASIC betöltőjét, és indítsuk el. Ha mindezzel készen vagyunk, a SYS 49152 paranccsal futtassuk a kódolót. Válaszoljunk a kérdéseire (kezdő- és végcím). Futtassuk le a dekódoló BASIC betöltőjét. Ha a dekódoló és a most már kódolt program egymás mögött helyezkedik el a tárban, akkor együtt is kimenthetjük a két programot.

3.3.5 Kódolás ASCII kódokkal

Az adatok kódolásának egy fantasztikusan ötletes módját fogjuk megismerni ebben a fejezetben. Kevés ilyen egyszerű és jól használható eljárás van, ami ráadásul feltűnően egyszerű. A kódolást most nemcsak egy új kódrendszer bevezetésével fogjuk tökéletesíteni, hanem az indítóprogramot egy igazán szokatlan helyre, a képernyőre fogjuk tölteni.

Az eljárás alapja az, hogy a gép az ASCII kódokat képernyőkódokká alakítja egy BSOUT nevű rutinnal – ezzel a rutinnal lehetőségünk van a programot az éppen aktuális képernyőterületre írni. A képernyő-memória áthelyezésével és a kurzor pozicionálásával szinte akárhová tehetjük a programot, amelyet a képernyőn kívánunk „elrejteni". A kurzor a pontos beállítást szolgálja. Csak nagyon kivételes képességű tolvajnak jut eszébe, hogy amit a képernyőn lát, az maga a program.

Miután az érdeklődést remélhetőleg kellőképpen felcsigáztuk, nézzük meg, hogyan is kell ezt a módszert alkalmazni. Az egyetlen nehézséget az okozza, hogy nem lehet minden ASCII kódot képernyőkóddá alakítani. Az ASCII kódtáblázatban szerepelnek olyan kódok, amelyek nem egy karakter kiírását jelentik, hanem egy feladat elvégzését. A vezérlőkarakterek nem íródnak ki a képernyőre; a hozzájuk tartozó művelet azonnal végrehajtódik. Az előbb tárgyalt vezérlőkarakter pl. a RETURN, melynek ASCII kódja a $0D (13). A másik probléma az, hogy a $7F-nél (127) nagyobb képernyőkódok jelentése ugyanaz, mint a $7F-fel kisebbeknek, de ezek inverz módban jelennek meg a képernyőn. A 6. táblázat ezekre mutat példát.

6. táblázat

ASCII kód	Inverz mód	Képernyőkód
$20	ki	$20
$20	be	$A0
$0D	akármi	vezérlőkarakter
RETURN		

Ahhoz, hogy a felmerült nehézségeket legyőzhessük, behatóan kell megvizsgálnunk a képernyő- és ASCII kódtáblázatot. A 7. táblázat mutatja, hogy az átváltás során csak a byte-ok alsó négy bitje változik meg.

7. táblázat

ASCII kód	Képernyőkód
$40–5F	$00–1F
$20	$20
$A0	$60
$B0	$70
$C0	$40
$D0	$50

A $7F-nél nagyobb kódokat más rendszer szerint alakítsuk át, mint az operációs rendszer teszi, ezzel egyértelmű negfeleltetést érünk el a képernyő- és ASCII kódok között, és ez egyféle „rejtjelezést" is jelenthet.

8. táblázat

ASCII kód	Képernyőkód
$E0	$80
$F0	$90
$60	$A0
$70	$B0
$80	$C0
$90	$D0
$00	$E0
$10	$F0

A 8. táblázatot érdemes jól megnézni, mert ez használják a fejezet végén ismertetett programok. A most előállított saját kódjainkat egy rutin segítségével visszaalakíthatjuk normál ASCII kódokká. Az inverz bit beállítása után már kiírhatjuk a képernyőre a kívánt adatokat.

A most következő program az eredményesebb kódolás érdekében még $19-cel meg is növeli minden byte értékét. A kódolandó byte-ot az akkumulátorban kéri, és az X regiszterben adja vissza a kódolt értéket.

```
1000 STX $FE       ;X reg. megőrzése
1002 STA $FD       ;A tárpolása
1004 AND #$F0      ; felső félbyte elkülönítése
1006 LDX #$0E      ;mutató a táblázatra
1008 CMP $101D,X   ;karakter összehasonl. a táblázattal
100B BEQ $1010     ;elágazás, ha egyeznek
100D DEX           ;mutató csökkentése
100E BPL $1008     ;ugrás
1010 LDA $FD       ;A visszatöltése
1012 AND #$0E      ;felső félbyte törlése
1014 ORA 102D,X    ;új félbyte
1017 CLC           ;carry törlése összeadáshoz
1018 ADC #$19      ;$19 hozzáadása
101A LDX $FE       ;X reg. visszatöltése
101C RTS           ;visszaugrás
101D BRK           ;
101E 10 20 30 40 50 60 70 80   ;ezekkel hasonlítja
1026 90 A0 B0 C0 D0 E0 F0 40   ;össze a byte-ot

102E 50 20 30 C0 D0 A0 B0 E0   ;a félbyte-ok
1036 F0 60 70 80 90 00 10 00
```

A kódolóhoz tartozó dekódoló rutin a következő:

```
1100 STX $FE       ;X reg megőrzése
1102 SEC           ;carry beállítása kivonáshoz
1103 SBC #$19      ;$19 kivonása
1105 STA $FD       ;az eredmény megőrzése
1107 AND #$F0      ;felső félbyte elkülönítése
1109 LDX #$07      ;mutato a táblázatra
110B CMP $1145,X   ;összehasonlítása a byte-tal
110E BEQ $1118     ;elágazás, ha egyeznek
1110 DEX           ;mutató csökkentése
1111 BPL $110B     ;elágazás, ha nem nulla
1113 LDA $FD       ;karakter bevitele
1115 CLC           ;carry törlése
1116 BCC $1123     ;ugrás
1118 LDA #$01      ;REVERSE flag
111A STA $C7       ;beállítása
111C LDA $FD       ;karakter bevitele
111E AND #$0F      ;alsó félbyte elkülönítése
1120 ORA $114D,X   ;és összehasonlítása az újjal
1123 JSR $FFD2     ;karakter kivitele
1126 LDA #$00      ;REVERSE flag
1128 STA $C7       ;törlése
112A LDX $FE       ;x reg visszaállítása
112C RTS           ;visszaugrás
112D NOP
112E NOP
```

A kódolás kezdőcímét az akkumulátorba és az X regiszterbe kell betenni (alsó, felső byte formában). A felhasználás megkönnyítésére közlünk egy programot, amely a kurzorpozicionálást végzi. Az akkumulátorba kell tenni annak a karakternek a kódját, amit ki akarunk írni. A rutin a 112F címen kezdődik.

```
112F STX $0288   ;a cím felső byte-ja
1132 LDX #$00    ;X reg. törlése osztáshoz
1134 CMP #$28    ;alsó byte
1136 BCC $113E   ;elosztása
1138 SEC         ;40-el
1139 SBC #$28    ;az osztás
113B INX         ;eredménye
113C DCS $1134   ;az X reg-ben van
113E STX $D6     ;
1140 STA $D3     ;
1142 JMP $E560   ;a cursor elhelyezése
1145 BRK
1146 10 60 70 80 90 E0 F0 A0  ;saját ASCII kódjaink táblázata

114E B0 20 30 C0 D0 40 50 00  ;normál kódba átszámításhoz
```

3.4 Tiltott kódok

A továbbiakban a C 64-es operációs rendszerének olyan kódjairól lesz szó, amelyeket a programozók ritkán használnak, mivel nem jelölnek egyetlen assembler utasítást sem; ezeket tiltott kódoknak fogjuk nevezni. (Ezekről a kódokról a különböző szakkönyvekben is csak elvétve olvashatunk. A lektor megjegyzése.)

3.4.1 A tiltott kódok jelentése

Aki már programozott gépi kódban, felfigyelhetett arra, hogy bizonyos kódok mellett nem assembler utasítások jelennek meg, hanem kérdőjelek. Gyakorta láthatjuk disassemblálás közben, hogy két assembler utasítás között több sorban kérdőjelek vannak:

```
C000 BPL $C12F
C003 ???
C004 ???
C005 ???
C006 EOR #$F2
```

A kérdőjelek magyarázata nagyon egyszerű. A C 64-es assembler utasításai nem egy byte-ból állnak. Az utasítások első byte-jai a tényleges műveletet jelölik a címzésmóddal együtt, a továbbiak pedig az utasítás paraméterét adják meg. A disassembler programok ezeket a kódokat felismerve alakítják vissza azokat assembler utasításokká. Ha nem az első byte-tól kezdjük egy utasítás disassemblálását, akkor a monitor rosszul fog dolgozni, pl. egy hárombyte-os utasítás második byte-ját fogja műveleti kódnak értelmezni, holott az a paraméter része. Mivel egy byte-nak 256 különféle értéke lehet, a C 64-es maximális utasításkészlete 256 utasításból állhat. Az utasítások számát korlátozza, hogy a különböző címzésmódokhoz más és más utasításkód tartozik; ám a címzésmódok korlátozásának ellenére még mindig maradnak olyan kódok, amelyek jelentése nem meghatározott. A tiltott kódok alatt ezeket értjük. Ha a rendszer egy tiltott kódhoz ér, akkor azt nem tudja lefordítani, és három kérdőjelet ír a képernyőre. Bár a 6510-es processzor nem használja ezeket a kódokat, ezeknek is van némi jelentésük. A tiltott kódokat három csoportra szokás felosztani. A 9.a) táblázat szemlélteti a tiltott kódok első csoportját.

9. a) táblázat

STOP	NOP 1. byte	NOP 2. byte	NOP 3. byte
02	1A	04	0C
12	3A	14	1C
22	5A	34	3C
32	7A	44	5C
42	DA	54	7C
52	FA	64	DC
62		74	FC
72		80	
92		82	
B2		89	
D2		C2	
F2		D4	
		E2	
		F4	

Ha a rendszer egy ilyen kódhoz ér, az vagy leblokkolja a gépet, vagy a $EA kódhoz hasonlóan (ami a NOP utasításnak felel meg) nem befolyásolja a további működést.

A tiltott kódok második csoportja (9.b) táblázat) a két ismert utasításnak felel meg. Amikor ezt kell a gépnek végrehajtania, akkor tulajdonképpen két utasítást hajt végre. Ezeket az utasításokat az STA-hoz hasonlóan lehet címezni.

9. b) táblázat

CÍM-ZÉS-MÓD	ASL: ORA	ROL: AND	LSR: EOR	ROR: ADC	DEC: CMP	INC: SBC	LDA: TAX
(,X)	03	23	43	63	C3	E3	A3
(),Y	13	33	53	73	D3	F3	B3
ABS	0F	2F	4F	6F	CF	EF	AF
ABS,X	1F	3F	5F	7F	DF	FF	/
ABS,Y	1B	3B	5B	7B	DB	FB	BF
ZP	07	27	47	67	C7	E7	A7
ZP,X	17	37	57	77	D7	F7	/
ZP,Y	/	/	/	/	/	/	B7

A harmadik csoportba (9.c) táblázat) tartozó kódok hasonlítanak a második csoport kódjaihoz, ezek is kettő vagy több utasítást hajtanak végre. A két vagy több funkció miatt ezeket az utasításokat igen nehéz beépíteni valamely programba.

9. c) táblázat

0B MM : AND MM :	negatív flag-et állít az átviteli bitbe
2B MM :	ua. mint az 0B MM
4B MM : AND MM :	LSR
6B MM :	ha decimális = 0; AND MM . ROR, ezenkívül az akku 0 bitjét átviszi az átviteli bitbe, az 5. és a 6. bit között EOR eredményét pedig a túlcsordulási bitbe
83 MM :	A AND X eredménye (MM,X)-be kerül
87 MM :	A AND X MM-be
8B MM :	TXA: AND MM
8F LO HI :	Az A AND eredménye LO, HI-be kerül
93 MM :	A AND X, majd ennek és MM + 1-nek az összege MM. X-be kerül
97 MM :	A AND X regiszter eredménye az MM, X-be kerül
9B LO HI :	A AND X a veremmutatóba, majd a veremmutató AND HI + 1 a LO HI, Y-ba kerül
9C LO HI :	Y AND HI + 1 HI LO, X-be kerül
9E LO HI :	X AND NN + 1 HI LO, Y-ba kerül
9F LO HI :	A AND X, HI + 1 a HI LO, Y-ba kerül
B B LO HI :	HI LO, Y AND veremmutató az x regiszterbe kerül, majd TXS:TXA
CBMM :	A AND X az X regiszterbe, majd x ninusz MM (átvitel nélkül)
EB MM :	SBC MM

3.4.2 A tiltott kódok alkalmazása

A tiltott műveleti kódok programbéli használata több szempontból is előnyös: így egy kóddal több utasítás végrehajtását is szimulálhatjuk; egyes utasításokat egyébként nem megengedett címzéssel használhatunk fel; és – ez talán a legfontosabb – közönséges módszerekkel nem lehet őket disassemblálni (nincs nekik megfelelő utasítás). Nem sokan kezdenek egy olyan, látszólag értelmetlen programrész visszafejtéséhez, amely tele van kérdőjellel. A felhasználás hátrányai között fel kell sorolni, hogy a program méreteit nem csökkenti jelentős mértékben az ilyenfajta programozás. Azért, hogy olyan címzésekkel használjunk utasításokat, ahogyan egyébként nem volna lehetséges, azzal kell fizetni, hogy a program tiltott kódokat tartalmazó részeit hexadecimális kódokkal kell beírni. Az ily módon titkosított programokat az, aki a tiltott kódok táblázatát nem ismeri, nem is tudja visszafejteni. A felsorolt hátrányok elég súlyosak ahhoz, hogy ne írjuk az egész programot ilyen titkosítással. A tiltott kódokat célszerű fejezetünk előző részeiben ismertetett különféle védelmek leginkább kiszolgáltatott részeiben felhasználni. Ha egy program dekódolóját ilyen módon írjuk, akkor nagyon megnehezítjük a tolvajok dolgát. További nehézségeket támaszthatunk azzal, ha azt a dekódolót is titkos kódok felhasználásával írjuk meg, amelyik az előzőt dekódolja. Ezek után biztos nem akad, aki „belepiszkál" a programunkba.

A tiltott kódok egy másik alkalmazása az, ha a programba sok-sok helyre NOP utasításoknak megfelelő műveleti kódokat építünk be, és ezzel zavarjuk meg a „betörőt". (Ezek a kódok az általunk ismertetett felosztás szerint az első csoportba tartoznak.)

A legjobb megoldás úgy érhető el, ha a programot a már ismert módon, tiltott kódokkal írjuk át.

A következő program ezt a módszert szemlélteti. A program a keret színét változtatja két egymásba ágyazott ciklusban. A belső ciklus változója a gép $FB (251) címen van, a külső ciklusé pedig az X regiszterben.

```
1000 LDX #$00   ;a külső ciklus mutatója
1002 LDA #$00   ;a belső ciklus értéke
1004 STA $FB    ;számláló beállítása
1006 DEC $D020  ;keretszín csökkentése
1009 DEC $FB    ;a mutató csökkentése
100B LDA $FB    ;a mutató bevitele
100D CMP #$00   ;vége van-e a belső ciklusnak
100F BNE $1006  ;ha nem akkor tovább
1011 INX        ;a külső ciklus mutatójának növelése
1012 BNE $1004  ;ha 256-nál kisebb akkor tovább
1024 BRK        ;program vége
```

Első lépésben a $1009; $100B és $100D címeken lévő három utasítást cseréljük ki. Az iménti három utasítás helyére a $C7 tiltott kódú utasítást írjuk. Ez az utasítás a mi felosztásunk szerinti második csoportba tartozik, egy DEC és egy CMP utasításnak felel meg, az utána lévő nulláslapú címzés pedig mind a kettőhöz tartozik.

```
1000 LDX #$00   ;a külső ciklus mutatója
1002 LDA #$00   ;a belső ciklusé
1004 STA $FB    ;ciklus beállítása
1006 DEC $D020  ;keretszín csökkentése
1009 ???        ;tiltott C7 és
100A ???        ;FB cím
100B BNE $1006  ;ciklus kezdetre
100D INX        ;mutató növelése
100E BNE $1006  ;cikluskezdetre
1010 BRK        ;program vége

1000 A2 00 A9 00 85 FB CE 20
1008 D0 C7 FB D0 F9 E8 D0 F6
1010 00 00 00 00 00 00 00 00
```

Második lépésben a $1000 és $1002-es sorokat helyettesítjük. A $AF utasítás az utána álló két byte tartalmát az akkumulátorba tölti, majd utána az akkumulátort áttölti az X regiszterbe. Nekünk arra van szükségünk, hogy mind az X regiszter, mind az akkumulátor tartalma nulla legyen, ezért a $E379 cím tartalmát töltjük be a regiszterekbe, ami nulla. A $E379 cím tartalma, mivel a ROM-ban van, mindig állandó. A módosítások elvégzése után a program a következő:

```
1000 ???          ;tiltott muveleti kód
1001 ADC $85E3,Y  ;LDA $E379 és
1004 ???          ;tax elvégzésére
1005 DEC $D020    ;keretszín csökkentése
1008 ???          ;tiltott kód
1009 ???          ;DEC $FB és CMP $FB elvégzésére
100A BNE $1005    ;ciklusKezdetre
100C INX          ;mutató növelése
100D BNE $1005    ;ciklusKezdetre
100F BRK          ;program vége

1000 AF 79 E3 85 FB CE 20 D0
1008 C7 FB D0 F9 E8 D0 F6 00
1010 00 00 00 00 00 00 00 00
```

A programlistán látható, hogy a STA FB utasítás is „eltűnt". Az STA utasítás azért nem szerepel a listán, mert a disassembler a $1000-es sorban lévő utasítást nem ismeri föl, helyette a $1001-es címen lévő kódot értelmezi. Ezért van a $1001 címen egy ADC utasítás. Az ADC utasításban szereplő címet az RTS utasítás kódjából alakítja ki a dissasembler program, és ezért nem történt az STA disassemblálása. A gép természetesen tudja értelmezni a $AF kódot, és nem esik abba a hibába, amibe a disassembler, a program futása tehát hibátlan lesz.

A program 1 byte-tal rövidebb lett, mivel a rejtett kódú utasítás kisebb helyet foglal el, mint az a kettő, ami helyett írtuk őket. (A program után közölt byte-sorozat figyelmes tanulmányozásával mindez kiderül.)

Az utolsó módosítás az, hogy a DEC $D020 utasítás helyett ismét egy $C7 tiltott kódú utasítást írunk. A CMP utasítás nem okoz problémát, mert utána is egy ilyen áll, és ezért a jelzőbitek állapota végül ennek megfelelően állítódik be.

```
1000 ???              ;
1001 ADC $85E3,Y      ;
1004 ???              ;
1005 ???              ;
1006 JSR $C7D0        ;
1009 ???              ;
100A BNE $1005        ;
100C INX              ;
100D BNE $1005        ;
100F BRK              ;programvég

1000 AF 79 E3 85 FB CF 20 D0
1008 C7 FB D0 F9 E8 D0 F6 00
1010 00 00 00 00 00 00 00 00
```

A JMP utasítás ugyanazért került a programba, mint az ADC.

3.4.3 A tiltott kódok ütemciklusa

A 10. táblázat a tiltott kódú utasítások ütemciklusait mutatja be.

10. táblázat

Hex	Ütem-cikl.	Hex	Ütem-cikl.	Hex	Ütem-cikl.	Hex	Ütem-cikl
03	8	4B	2	8F	2	E3	8
04	3	4F	6	93	6	E7	5
07	5	53	8	97	4	EB	2
0B	2	54	4	9B	5	EF	6
0C	4	57	6	9C	5	F3	8
0F	6	5A	2	9E	5	F4	4
13	8	5B	7	9F	5	F7	6
14	4	5C	5	A3	6	FA	2
17	6	5F	7	A7	3	FB	7
1A	2	63	8	AF	4	FC	5
1B	7	64	3	B3	5	FF	7
1C	5	67	5	B7	4		

10. táblázat

Hex	Ütem-cikl.	Hex	Ütem-cikl.	Hex	Ütem-cikl.	Hex	Ütem-cikl.
1F	7	6B	2	BB	5		
23	8	6F	6	BF	5		
27	5	73	8	C2	2		
2B	2	74	4	C3	8		
2F	6	77	6	C7	5		
33	8	7A	2	CB	2		
34	4	7B	7	CF	6		
37	6	7C	5	D3	8		
3A	2	7F	7	D4	4		
3B	7	80	2	D7	6		
3C	5	82	2	DA	2		
3F	7	83	6	DB	7		
43	8	87	3	DC	5		
44	3	89	2	DF	7		
47	5	8B	2	E2	2		

3.5 A dongle mint programvédelmi eszköz

Idáig kizárólag szoftver úton próbáltuk megvédeni programjainkat. Ebben az alfejezetben viszont a szoftvereljárások mellett egy kisebb hardvereszközt is fogunk használni. A dongle-t nemigen szokták használni azok, akik csak otthon hobbyból programoznak, még komolyabb szoftverházak is elég ritkán használják. A dongle egy olyan egyszerű berendezés, amely olcsón előállítható, és különösebb szaktudást sem igényel.

A védelmek ezen módjáról állítható csak, hogy igazán jó hatásfokkal védi meg a programokat. Aki ezt az eszközt nem vásárolta meg, az nem fogja tudni az ezzel védett programokat futtatni. Természetesen ez is elkészíthető otthon, de a dongle másolását nagyon megnehezíthetjük. A dongle használata lehetővé teszi, hogy a tökéletes másolatokat se tudja az használni, akinek nincs megfelelő dongle-ja. Az ilyen módon védett lemezhez akár egy másolóprogramot is mellékelhetünk, a hardvereszköz nélkül úgysem lehet használni a programot.

A tolvajok egyetlen lehetősége ezzel az eljárással szemben, hogy a programhoz tartozó dongle-t is lemásolják. A másolást különböző módokon megnehezíthetjük, pl. egybeöntjük a dongle-t a dobozával, és így nem lehet hozzáférni

a kapcsolási rajzához. Ne lehessen bemérni, hogy mi van a dobozban. Minél bonyolultabb hardvert alkalmazunk, annak egy apró hibája is elegendő lehet a program futásának megakadályozásához.

3.5.1 Hogyan működik a dongle?

A legegyszerűbb dongle-k működése a kontrollporton alapul. A kontrollporthoz kell csatlakoztatni, és programból folyamatosan (a program több helyén) vizsgálni, hogy valóban a porton van-e. Ha a dongle nincs a helyén, akkor a rendszer valamiben eltér, ezt az eltérést vizsgálva a program futását megszakíthatjuk. A vizsgálatot a már ismertetett programtitkosítási eszközökkel leplezhetjük.

A legegyszerűbb dongle a vezérlőport néhány vezetékét földeli le, azaz a gép néhány bitjét alacsony, nulla szintre állítja. A dongle működésének megértéséhez a kontrollport működéséről is kell tudnunk néhány dolgot. A most következők mind a két portra igazak, de az általunk leírtak a 2-es protra értendők. A vezérlőport 0-tól 4-ig terjedő vezetékei alapállapotban magas, logikai egyes szinten vannak. A $DC00 (a CIA chip 0. regisztere) címen a 0-tól 4-ig terjedő bitek a kontrollport 0-tól 4-ig számozott lábainak jelszintével egyezik meg (ha nem csatlakozik semmi, akkor 1). A kontrollport 7. vezetéke a földet jelenti, azaz nulla szinten van.

A $DC00 (56320) cím bitjeit a vezérlőport megfelelő lábainak a 7. lábra kötésével állíthatjuk alacsony szintre. A 0-tól 3-ig terjedő bitek a botkormány négy irányát jelölik, a 4. bit pedig a tűzgombot. Aktív állapotban mindegyik nulla. Ez a fajta dongle tulajdonképpen nem más, mint néhány vezeték földre kötése. A dongle ellenőrzését $DC00 cím lekérdezésével végezhetjük el. Ha a dongle-t egy darabba öntjük a „házával", akkor nem lehet lemásolni.

3.5.2 Egyszerű lekérdezés

Ebben a példában a 0-s és 1-es vezetéket kötjük a földre. A 0. bit a FEL, az 1-es pedig a LE irányt mutatja. Azért ezt a kettőt választottuk, mert ezeket egyetlen botkormánnyal nem lehet szimulálni (hisz nem lehet egyszerre LE és FEL húzni a kart). A dongle egy vezérlőport-csatlakozót és három huzaldarabot tartalmaz – ezek az alkatrészek minden elektronikai szaküzletben nagy mennyiségben megtalálhatók.

Most nézzük a dongle lekérdezéséhez szükséges programot.

```
10 A$="{CLR}O.K."
20 PRINT"{CLR} KEREM CSATLAKOZTASSA A DONGLE-T ES"
25 PRINT"USSON LE EGY BILLENTYUT"
```

```
30 POKE198,0:WAIT 198,1
40 A=PEEK(56320) AND 3
50 IF A<>0 THEN GOTO70
60 PRINT A$:GOTO40
70 PRINT"VEGE",A
```

A billentyűzet lekérdezése a kontrollport regiszterén keresztül valósul meg, ezért a billentyűzet lekérdezése a továbbiakban nem folytatódik.

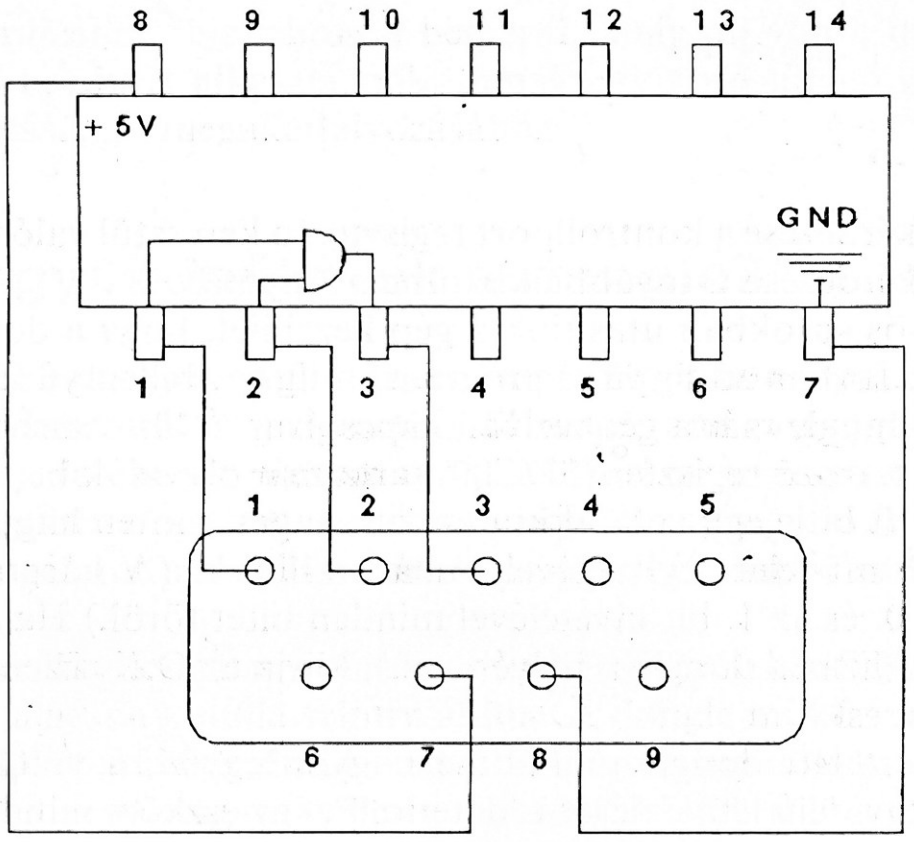
A 20-as és 25-ös sorokban utasítjuk a gép kezelőjét, hogy a dongle-t csatlakoztassa. A 30-as sorban addig vár a program, amíg egy billentyű leütésével nem jelezzük, hogy a dongle már a géphez van kapcsolva. A 40-es sorban a második botkormányhoz tartozó regiszter ($DC00) tartalmát olvassuk be. Ha a beolvasott byte 0. vagy 1. bitje egy volt, akkor ezeket magas szinten hagyjuk, a többit viszont az AND művelet segítségével nullára állítjuk. (A hárommal végzett AND művelet a 0. és az 1. bit kivételével minden bitet töröl.) Ha az A változó értéke nem nulla (azaz a dongle a helyén van), kiírja az O.K. üzenetet, és kezdi elölről az ellenőrzést.

Az imént ismertetett dongle működése nagyon egyszerű volt. A működés néhány vezeték egyidejű leföldelését jelentette. Ezt az eszközt minden szaktudás nélkül is akárki el tudja készíteni; sem különösebb gyakorlatot, sem speciális szerszámokat nem igényel, másolása tehát nem kerül túl nagy erőfeszítésbe. A C 64-tulajdonosok többsége azonban mégsem vállalkozna egy ilyen hardvereszköz elkészítésére.

3.5.3. Dongle IC-vel

Az előbbinél jobb védekezés, ha nem mindig ugyanazt a vezetéket földeljük, hanem (valamilyen rendszer szerint) mindig mást. A vezetékek leföldelésének változtatását egy IC-vel érjük el. Az IC típusa nem lényeges, számunkra csak az fontos, hogy különböző logikai kapukból álljon. Mivel a billentyűzet lekérdezése a $DC00 címen keresztül folyik, ezért lehetőleg a vezetékeket a lehető legtöbbet magasan kell tartani, hogy amikor az operációs rendszer fontos számára, a biteket le tudja „húzni" nullára. A NAND művelet nem használható, mert ennél a műveletnél legalább egy vezeték logikailag alacsony szinten van. A következő ábrán négy egymástól független *ÉS* kapuval rendelkező IC-t alkalmazunk, de a kapuk közül csak egyet használunk fel. Az ÉS kapunak két bemenete van, ami alaphelyzetben magas szinten áll.

A 3. ábra után közöljük azt a rutint, ami a dongle ellenőrzését végzi. Az ellenőrzés teljes egészében megszakításból történik, így a főprogram futását nem befolyásolja, az zavartalanul működhet. Ha nem jó dongle van a rendszerben (vagy esetleg egyáltalán nincs), akkor a rutin egy RESET-et generál. A RESET helyett természetesen saját belátásunk szerint mást is alkalmazhatunk. A RESET-et pl. egy olyan rutinra cserélhetjük fel, ami törölheti a tárat, vagy egysze-



rűen kiírhat egy olyan feliratot, ami addig vár, amíg a megfelelő dongle nincs a gépben. Az ellenőrző rutinok védelmére különböző, az előző alfejezetekben ismertetett eljárásokat lehet alkalmazni. A 3.4 alfejezet tiltott kódjai nyújtják a legbiztosabb védelmet azok ellen, akik megpróbálják az ellenőrző rutint kivenni a programból.

A megszakítási vektoroknak az ellenőrző rutinra való állítása:

```
C000 SEI          ;megszakítás letiltása
C001 LDA #$0D     ;az új vektor alsó byte-jának
C003 STA $0314    ;tárolása
C006 LDA #$C0     ;a felső byte
C008 STA $0315    ;
C00B CLI          ;a megszakítás engedélyezése
C00C RTS          ;visszaugrás
```

Az ellenőrző rutin:

```
C00D PHA          ;A a verembe
C00E LDA $DC00    ;2-es port értéke
C011 PHA          ;a verembe
C012 AND #$FE     ;0. bit törlése
C014 STA $DC00    ;visszaírás
C017 LDA $DC00    ;új érték
C01A AND #$05     ;0 és 2 bit elkülönítése
C01C BNE $C037    ;beállítás után RESET
C01E PLA          ;a port értékének
C01F STA $DC00    ;visszaírása
C022 LDA $DC00    ;új érték
C025 ORA #$01     ;0 bit beállítása
```

```
C027 STA $DC00   ;visszaírása
C02A LDA $DC00   ;új érték bevitele
C02D AND #$0     ;a 0 és 2 bit elkülonítése
C02F EOR #$0     ;bitek invertálása
C031 BNE $C037   ;ha nincsenek beállitva RESET
C033 PLA         ;A a veremből
C034 JMP $EA31   ;a megszakitásrutinra
```

A RESET végrehajtása nem megfelelő dongle esetén:

```
C037 INC $D020   ;keretszin növelese
C03A SEI         ;a megszakitás letiltása
C03D JMP $FCEF   ;CM80 v. modul lekérd. nélkül RESET
```

Akik nem rendelkeznek monitorprogrammal, azok számára közöljük a program BASIC betöltőjét:

```
10 FORI=0TO61
20 READA$:A$=MID$(A$,1,2):B$=RIGHT$(A$,1):C$=LEFT$(A$,1)
30 B=ASC(B$)-48:IFB>9THENB=B-7
40 C=ASC(C$)-48:IFC>9THENC=C-7
50 D=16*C+B:POKE28672+I,D
60 NEXT
100 DATA 78        :REM          SEI
110 DATA A9,0D     :REM          LDA #$0D
120 DATA 8D,14,03:REM            STA $0314
130 DATA A9,C0     :REM          LDA #$C0
140 DATA 8D,15,03:REM            STA $0315
150 DATA 58        :REM          CLI
160 DATA 60        :REM          RTS
170 DATA 48        :REM          PHA
180 DATA AD,00,DC:REM            LDA $DC00
190 DATA 48        :REM          PHA
200 DATA 29,FE     :REM          AND #$FE
210 DATA 8D,00,DC:REM            STA $DC00
220 DATA AD,00,DC:REM            LDA $DC00
230 DATA 29,05     :REM          AND #$05
240 DATA D0,19     :REM          BNE C1
250 DATA 68        :REM          PLA
260 DATA 8D,00,DC:REM            STA $DC00
270 DATA AD,00,DC:REM            LDA $DC00
280 DATA 09,01     :REM          ORA #$01
290 DATA 8D,00,DC:REM            STA $DC00
300 DATA AD,00,DC:REM            LDA $DC00
310 DATA 29,05     :REM          AND #$05
320 DATA 49,05     :REM          EOR #$05
330 DATA D0,04     :REM          BNE C1
340 DATA 68        :REM          PLA
350 DATA 4C,31,EA:REM            JMP $EA31
360 DATA EE,20,D0:REM C1         INC $D020
370 DATA 78        :REM          SEI
380 DATA 4C,EF,FC:REM            JMP $FCEF

READY.
```

A bemutatott megoldások természetesen messze nem foglalják magukban az összes lehetőséget. Az iménti eljárás csak „kóstoló" volt a hardvervédelmekből. Használhatjuk pl. a user-portot is védelmi célra, a kontrollport nem az egyedüli lehetőség.

3.6 Jelszólekérdezés

A jelszó lekérdezésének a célja, hogy programunkhoz csak a jelszó ismeretében lehessen hozzáférni. A jelszólekérdezés technikáját alkalmazhatjuk adathalmazok megvédésére is. Nem tanácsos a jelszót a program végére vagy elejére tenni, mert ezzel nagyon megkönnyítjük azok dolgát, akik valamilyen módon a jelszó megkeresésével próbálkoznak. A jelszólekérdezést BASIC-ből és gépi kódból egyaránt használhatjuk. BASIC programokban természetesen csak akkor, ha letiltjuk a LIST parancsot. A LIST parancs hatástalanítása nélkül a védelem teljesen hasznavehetetlen.

A jelszót lekérdező program egy változata:

```
10 POKE808,225:PRINT CHR$(147):A=0
20 POKE19,1
30 PRINT CHR$(19): INPUT"JELSZO :";A$
40 POKE 19,0:PRINT CHR$(13)
50 IF A$=""THEN 20
60 IF A$="PETI" THEN PRINT CHR$(17);" ELFOGADOM":END
70 GOTO20

READY.
```

A 20-as sor POKE utasítása az INPUT kérdőjelének letiltását végzi. Ha a felhasználó háromszor elrontja a jelszót, a program a RESET rutint hívja meg. Az A változó az eredménytelen kísérleteket számolja. Ha nem írunk jelszót, a program nem számol hibát. A példában szereplő jelszó a "PETI". Ha a program ezt a jelszót kapja, kiírja az "ELFOGADOM" üzenetet, majd a program megáll. Miután beépítettük ezt a rutint a főprogramba, az END helyett természetesen a főprogramba való visszatérést kell írni.

Mivel a LIST letiltása ellenére monitorprogramból könnyen módosíthatnánk a programot, érdemes ezt a részt gépi kódban megírni, és azt titkosítani. Nézzünk erre egy példát!

```
C000 JSR $E544    ;a képernyő törlése
C003 LDX #$00     ;a számláló nullára állítása
C005 LDA $C063,X  ;jelszó
C008 JSR $FFD2    ;kiírása
C00B INX          ;a számláló növelése
C00C CPX #$09     ;ha minden betűt betöltöttünk
```

```
C00E BNE $C005    ;tovább
C010 LDX #$00     ;a számláló nullára állítása
C012 JSR $FFCF    ;elágazás az INPUT rutinra
C015 STA $C080,X  ;a beolvasott bytebtárolása
C018 INX          ; a számlaló növelese
C019 CMP #$0D     ;várakozás a RETURN-re
C01B BNE $C012    ;ha leütötték, tovább
C01D DEX          ;a RETURN kód törlése
C01E STX $02      ;a számláló tárolása
C020 LDX #$00     ;a számláló nullára állítasa
C022 LDA $C06C,X  ;a jelszó szövegének
C025 EOR #$24     ;dekódolása
C027 STA $C06C,X  ;és tárolása
C02A INX          ;számlaló növelese
C02B CPX #$06     ;ha minden dekódolva,
C02D BNE $C022    ;akkor tovább
C02F LDX #$00     ;a számláló nullára állítása
C031 LDA $C080,X  ;a beolvasott jellszó
C034 CMP $C06C,X  ;összehasonlítása
C037 BE0 $C03C    ;ha egyezik, akkor tovább,
C039 JMP $C051    ;egyébkent a dekódolás és RESET
C03C INX          ;a számláló növelése
C03D CPX $02      ;a hosszak összehasonlítása
C03F BNE $C031    ;ha az összes ellenőrizve, tovább
C041 LDX #$00     ;a számláló nullára állítádsa
C043 LDA $C06C,X  ;jelszó
C046 EOR #$24     ;ismételt dekódolása
C048 STA$C06C,X   ;és tárolása
C04B INX          ;a számláló növelése
C04C CPX #$06     ;ha minden dekódolva, akkor
C04E BNE $C043    ;tovább
C050 RTS          ;visszaugrás
C051 LDX #$00     ;a számláló nullára állítása
C053 LDA $C06C,X  ;jelszó
C056 EOR #$24     ;dekódolása
C058 STA $C060,X  ;és tárolása
C05B INX          ;növelése
C05C CPX #$06     ;ha minden dekódolva,
C05E BNE $C053    ;akkor tovább
C060 JMP $FCE2    ;rendszer RESET
C063 4A 45 4C 53 5A 4F 20 20 JELSZO
C06B 3A 00 00 00 00 00 00 00 :.......
```

A gépi kódú rutint BASIC programból a SYS49152 utasítással indíthatjuk el. Ez a gépi kódú program gyakorlatilag ugyanazt teszi, mint a BASIC. A jelszót a $C06 címtől kezdve helyeztük el. A jelszó egyetlen korlátozása, hogy maximum 16 karakter hosszú lehet. A jelszó beírása közben figyelnünk kell arra, hogy a kódoláshoz egy EOR #$24 műveletet kell végeznünk. A jelszó módosításának első lépése, hogy a jelszó ASCII karaktereit C06C-től kezdve tároljuk. A második lépés, hogy a szöveg hosszát be kell írnunk a $C02B, $C04C és $C05C címekre. Harmadik lépésben ugorjunk a $C051 címre, ez elvégzi a kódolást, majd egy RESET-et vált ki. Ezzel a módosítást befejeztük.

```
10 FORI=0TO114
20 READA$:A$=MID$(A$,1,2):B$=RIGHT$(A$,1):C$=LEFT$(A$,1)
30 B=ASC(B$)-48:IFB>9THENB=B-7
40 C=ASC(C$)-48:IFC>9THENC=C-7
50 D=16*C+B:POKE28672+I,D
60 NEXT
100 DATA 20,44,E5:REM          JSR $E544
110 DATA A2,00   :REM          LDX #$00
120 DATA BD,63,C0:REM C1       LDA $C063,X
130 DATA 20,D2,FF:REM          JSR $FFD2
140 DATA E8      :REM          INX
150 DATA E0,09   :REM          CPX #$09
160 DATA D0,F5   :REM          BNE C1
170 DATA A2,00   :REM          LDX #$00
180 DATA 20,CF,FF:REM C2       JSR $FFCF
190 DATA 9D,80,C0:REM          STA $C080,X
200 DATA E8      :REM          INX
210 DATA C9,0D   :REM          CMP #$0D
220 DATA D0,F5   :REM          BNE C2
230 DATA CA      :REM          DEX
240 DATA 86,02   :REM          STX $02
250 DATA A2,00   :REM          LDX #$00
260 DATA BD,6C,C0:REM C3       LDA $C06C,X
270 DATA 49,24   :REM          EOR #$24
280 DATA 9D,6C,C0:REM          STA $C06C,X
290 DATA E8      :REM          INX
300 DATA E0,06   :REM          CPX #$06
310 DATA D0,F3   :REM          BNE C3
320 DATA A2,00   :REM          LDX #$00
330 DATA BD,80,C0:REM C5       LDA $C080,X
340 DATA DD,6C,C0:REM          CMP $C06C,X
350 DATA F0,03   :REM          BEQ C4
360 DATA 4C,51,C0:REM          JMP L1
370 DATA E8      :REM C4       INX
380 DATA E4,02   :REM          CPX $02
390 DATA D0,F0   :REM          BNE C5
400 DATA A2,00   :REM          LDX #$00
410 DATA BD,6C,C0:REM C6       LDA $C06C,X
420 DATA 49,24   :REM          EOR #$24
430 DATA 9D,6C,C0:REM          STA $C06C,X
440 DATA E8      :REM          INX
450 DATA E0,06   :REM          CPX #$06
460 DATA D0,F3   :REM          BNE C6
470 DATA 60      :REM          RTS
480 DATA A2,00   :REM L1       LDX #$00
490 DATA BD,6C,C0:REM C7       LDA $C06C,X
500 DATA 49,24   :REM          EOR #$24
510 DATA 9D,6C,C0:REM          STA $C06C,X
520 DATA E8      :REM          INX
530 DATA E0,06   :REM          CPX #$06
540 DATA D0,F3   :REM          BNE C7
550 DATA 4C,E2,FC:REM          JMP $FCE2
560 DATA 4A,A5,4C:REM  JEL
570 DATA 53,5A,4F:REM  SZO
580 DATA 20,20,3A:REM    :
590 DATA 00,00,00:REM  ...
600 DATA 00,00,00:REM  ...
610 DATA 00      :REM  .

READY.
```

4. A LEMEZEK MÁSOLÁS ELLENI VÉDELME

Miben áll a lemezek másolásvédelmének lényege? Hogyan tudjuk alkalmazni a másolásvédelmi módszereket? Ezekre a kérdésekre kívánunk választ adni ebben a fejezetben. A bemutatott ötletek a másolásvédelmi módszereken túl, önmagukban is érdekesek, mint a lemezegység programozási lehetőségei.

Arra kérdésre, hogy mi is a másolásvédelem, általánosságban azt mondhatjuk: a lemezen végzett manipuláció, amely megakadályozza, hogy a lemezt valaki a közhasználatú másolóprogramok valamelyikével sokszorosítsa.

A programok tökéletes megértése nem nélkülözheti a lemezegység működésének alapos ismeretét, a gépi kódú programozást, jóllehet ezen ismeretek közlésére a könyv keretei között nem kerülhetett sor. Hogy mégis használható anyagot nyújtsunk azoknak is, akik a jelzett témakörökben nem elég jártasak, úgy készítettük el a programokat, hogy csak a legszükségesebb részeket írtuk gépi kódban, s ezeket a részeket DATA sorok formájában adjuk közre. Mielőtt fejest ugranánk az első másolásvédelmi módszer tárgyalásába, ismerkedjünk meg a lemezegység „lelkivilágával".

4.1 A DOS működése

A VC1541-es „intelligens" periféria, saját mikroprocesszorral és operációs rendszerrel (Disk Operating System, DOS). A lemezegység tehát nem veszi igénybe a számítógép központi tárát és munkaidejét; önállóan végrehajtja a számítógéptől kapott parancsokat.

A lemezegységnek éppen ezért háromféle tevékenységet kell párhuzamosan folytatnia. Le kell bonyolítania a géptől jövő és a gép felé irányuló adatforgalmat. Értelmeznie kell és végre kell hajtania az adatállományokra, a hozzá tartozó átviteli csatornára és blokkpufferre vonatkozó parancsokat. Végül el kell végeznie a hardveres lemezműveleteket, mint pl. egy blokk felírása, egy

blokk olvasása és a lemez formázása. Ezeket a feladatokat a VC1541-es lemezegység egy 6502-es mikroprocesszor segítségével végzi. A megoldást a megszakítási technika jelenti, lehetővé téve három program látszólag egyidejű futtatását.

A lemezpufferben elhelyezett saját gépi kódú programokban természetesen hivatkozhatunk a DOS rutinokra. Jóllehet a DOS az összes lemezkezelő rutint készen tartalmazza, a DOS rutinok sajátos szerkezete miatt használatuk nem olyan egyszerű, mint a számítógép rendszerrutinjai esetében. A lemezegység operációs rendszere két részre osztható. Az első rész a főprogram, amely állandóan fut. Ez a program kezeli pl. a soros buszt. A rutinok többségére rögzített címekkel hivatkozik, és azokból JMP utasítással tér vissza. Ez a magyarázata annak, hogy saját programjainkból ezekre a rutinokra nem hivatkozhatunk; csak úgy, ha részfeladatokra bontva, szintén saját készítésű rutinokkal oldjuk meg.

Az operációs rendszer másik része tulajdonképpen egy megszakítási program, ún. jobciklusokkal. Feladata az író/olvasó műveletek lebonyolítása. A rendszer minden megszakítási lépésben beolvassa a nulláslapon a $00-tól $05-ig terjedő címek tartalmát, amely a főprogram és a megszakítási program közötti kommunikáció adatterülete. A $80-nál (128-nál) nagyobb értékeket a program utasításnak (jobnak) tekinti, és végrehajtja. Minden jobcím egy meghatározott tárterületre hivatkozik, ahol a művelethez tartozó gépi kódot elhelyezték. Azt, hogy a művelet mely lemezterületre vonatkozik, a jobtárhoz rendelt további két cím tartalma (sáv, szektor) határozza meg. A 11. táblázat az egyes jobokhoz tartozó sáv–szektor címet, ill. tárterületet foglalja össze.

11. táblázat

Puffer	Cím	Job	Sáv	Szektor
	$0000–$00FF	Nem elérhető (nulláslap)		
	$0100–$01FF	Nem elérhető (verem)		
	$0200–$03FF	Nem elérhető (parancspuffer)		
#0	$0300–$03FF	$0000	$0006	$0007
#1	$0400–$04FF	$0001	$0008	$0009
#2	$0500–$05FF	$0002	$000A	$000B
#3	$0600–$06FF	$0003	$000C	$000D
	$0700–$07FF	Nem elérhető (BAM)		

A jobkódok és jelentésük:

$80 (128) Szektor olvasása (READ).
$90 (144) Szektor írása (WRITE).
$A0 (160) Szektor ellenőrzése (VERIFY).
$B0 (176) Szektor keresése (SEEK).
$C0 (192) A fej beállítása (BUMP).

$D0 (208) A pufferben elhelyezett program végrehajtása (JUMP).
$E0 (224) A pufferbeli program végrehajtása, a motor bekapcsolása és a fej pozicionálása után (EXECUTE).

A jobkódok feldolgozásának módját egy példán szemléltetjük. Ha egy blokkot BASIC-ből, az U1 utasítás alkalmazásával olvasunk be, a főprogram átveszi és azonosítja az utasítást, majd a blokk beolvasását engedélyezi. Az engedélyezés során a sáv-, ill. szektorszámát beírja arra a tárcímre, amely a kijelölt pufferhez tartozik, majd a job kódját ($80-at) betölti a jobtárba.

Tegyük fel, hogy a blokk beolvasásához a 2-es puffer tartozik, a sáv-, szektorértéke pedig a $0A, $0B címeken található. Tárolás után a rendszer elindítja a jobciklust. A jobciklusban ellenőrzi a jobtárat (ahol most a $80 kód áll), majd betölti a sáv- és szektorértékeket a $0A, $0B címekről. Ezután a fejet a kijelölt blokkra pozicionálja, és a blokk tartalmát beolvassa. A beolvasott adatok a 2-es pufferbe ($0500–$05FF) kerülnek. A főprogram ez idő alatt „várakozik". A várakozó ciklusból mindaddig nem lép ki, amíg a jobok "READY" üzenete meg nem érkezik.

Az üzenet szintén a jobtárba kerül, ahol előzőleg az utasítás volt. A visszajelzést a főprogram meg tudja különböztetni a jobkódoktól, ui. a jobkódok legmagasabb helyiértékű bitje 1, az üzeneteké pedig 0. A főprogram tehát a legmagasabb helyiértékű bit alapján eldönti, hogy a tárbeli adat jobkód vagy üzenet, és ha az utóbbi, a kiértékelést megkezdi.

A 12. táblázatban összefoglaljuk az üzenetkódokat a hozzájuk tartozó jelentésükkel együtt. (A hibaüzenetekkel a későbbiekben még bővebben foglalkozunk.)

12. táblázat

A DOS hibajelzései

Kód	Jelentés	DOS hibaüzenet
$01	Minden rendben	00, OK
$02	A fejblokk (header) nincs meg	20, READ ERROR
$03	A SYNC (szinkronjel) nincs meg	21, READ ERROR
$04	Az adatblokk nincs meg	22, READ ERROR
$05	Hiba az ellenőrző összegben	23, READ ERROR
$07	Verify-hiba	25, WRIRE ERROR
$08	A lemez írásvédett	26, WRITE PROTECT ON
$09	Hibás ellenőrző összeg a fejblokkban	27, READ ERROR
$0B	Rossz az ID	29, ID MISMATCH
$0F	Nincs lemez az egységben	74, DRIVE NOT READY

A lemezvezérlő rutinok alkalmazásához a következőket kell tudni: amikor egy blokkot U1 paranccsal beolvasunk, a vezérlő megkapja a »blokk olvasása« parancsot. Ha a vezérlő hibát jelez, a logikai DOS rutinok (a főprogramban) megpróbálnak a kijelölt sávtól fél szektorral jobbra, ill. balra adatot olvasni. Ha ez is eredménytelen, a rendszer „kiakasztja" az olvasófejet (a fejet az 1-es sávra állítja, és az olvasási hibára jellemző zajt kelt), majd az egész műveletet megpróbálja megismételni. A DOS általában öt olvasási kísérletet végez, mielőtt hibát jelez. A lemezvezérlő rutinok közvetlen hívásával ezt az eljárást tökéletesen kikerüljük. Az író/olvasó műveleteket ez esetben az U1, U2 parancsokkal végezhetjük el.

4.1.1 Az írás/olvasás programozási technikája

Ismerkedjünk meg közelebbről az adatok tárolásának és visszaolvasásának technikájával is! A lemezegység vezérlésének legfontosabb regiszterei a VIA 1-es és 2-es regisztere. Ez a két regiszter a $1C00, ill. $1C01 címeken található, és míg a rendszer a $1C00-ás címet a LED és a motor vezérléséhez adja, addig a $1C01 címet az olvasás/írás műveletek legbonyolítása közben használja.

A $1C00 cím egyes bitjeinek jelentése:

0. bit: A léptetőmotor vezérlése
1. bit: A léptetőmotor vezérlése
2. bit: A lemezegység motorja; 0 = a motor kikapcsolva
3. bit: A lemezegység LED-je; 0 = a LED kikapcsolva
4. bit: Írásvédelem; 0 = fénysorompó fel
5. bit: A bitsebesség beállítása
6. bit: Ua., mint az 5. bit
7. bit: A SYNC jel azonosítása olvasásnál; 0 = a SYNC megvan

Mint már említettük, az adatok írása/olvasása közben a rendszer a $1C01 címet használja. A rendszer azt az adatot írja fel a lemezre, amit előzőleg itt tároltunk, feltéve, hogy előzőleg felíró műveletet kértünk. De vajon honnan tudja meg a programozó azt, hogy az adott pillanatban a felírás/beolvasás befejeződött vagy sem?

Erre szolgál az ún. BYTE–READY vezeték, amely a processzor állapotregiszterének OVERFLOW kapcsolójával van összekötve. A BYTE–READY vezeték ezt a kapcsolót magasra állítja, és utólag nem törli. A törlést a programozónak „kézzel", a CLV utasítás segítségével kell elvégeznie, hogy a kapcsolót az állandó "Ready" állapotból kibillentse. Az egy byte felírásához szükséges időtartam 20 mikroszekundum körül mozog, a sebességkapcsoló beállításától függően.

A lemezprogramok elindítására kétféle lehetőség kínálkozik. A programot vagy a $E0 és $D0 jobkódokkal megszakítási rutinként, vagy közvetlen paranccsal indíthatjuk el. Elsőként a jobkódokra épített indítási módszert tárgyaljuk. Nézzünk egy példát egy byte beolvasásának programozására:

```
C1  BVC  C1
    CLV
    LDA  $1C01
```

4.1.2 DOS védelmi módszerek

4.1.2.1 A Commodore dekódoló rendszere

A védelmi rendszerek megértésének alapvető feltétele az, hogy pontosan tudjuk, hogyan rögzíti egy szektor tartalmát a lemezen a 1541-es. Minden szektor két részből – a blokkfejből és az adatblokkból – épül fel. Vegyük szemügyre közelebbről az egyes részek belső szerkezetét.

A blokkfej (16 nyolcbites byte) felépítése:

12/a táblázat

A byte-ok	
száma	tartalma
–	Szinkronjel
1	A blokkfej azonosítója ($08)
1	A blokkfej ellenőrző összege
1	A szektor sorszáma
1	A sáv sorszáma
1	ID LO (lemezazonosító alsó byte)
1	ID HI (lemezazonosító felső byte)
2	A blokkfej vége ($0F)
8	Blokkfej gap ($55)

Az adatblokk (260 nyolcbites byte) felépítése:

13. táblázat

A byte-ok	
száma	tartalma
–	Szinkronjel
1	Az adatblokk azonosítója ($07)
256	Adatbyte-ok
1	Az adatblokk ellenőrző összege
2	Az adatblokk vége ($00)
változó	Gap ($55)

A 1541-es a lemez felületére egyetlen folytonos bitsorozatként írja fel. A szektorok kezdetét és végét a sávon belül sehol sem jelzi demagnetizált zóna. A Commodore a szektorokat a szinkronjelek alapján különbözteti meg egymástól. A DOS 2.6 szinkronjele öt egymást követő $FF tartalmú nyolcbites byte, minden blokkfej és adatblokk elülső végére felírva. Az adatblokk és a szinkronjel megkülönböztetése a 1541-es kétféle üzemmódján alapul; a 1541-es vagy normál üzemmódban, vagy szinkronmódban ír a lemezre. Normál üzemmódban írás előtt a rendszer minden nyolcbites byte-ot átkódol tízbites alakra. A kódolás a bináris alaknak az ún. GCR (Group Code Recording) alakot felelteti meg. A konverzió rendkívül egyszerű: a nyolcbites byte-ból először két félbyte – alsó és felső félbyte – lesz. Matematikailag minden négybites félbyte megfeleltethető egy 0 és 15 (minden bit alacsony, ill. minden bit magas) közé eső hexadecimális számjegynek. A 1541-es GCR kódtáblázatában tehát mindössze a következő 16 megfeleltetésnek kell szerepelnie.

14. táblázat

Eredeti adat		GCR kód	Eredeti adat		GCR kód
$0	0000	01010	$8	1000	01001
$1	0001	01011	$9	1001	11001
$2	0010	10010	$A	1010	11010
$3	0011	10011	$B	1011	11011
$4	0100	01110	$C	1100	01101
$5	0101	01111	$D	1101	11101
$6	0110	10110	$E	1110	11110
$7	0111	10111	$F	1111	10101

A táblázat alapján kövessük a $12 (18) GCR konverzióját.

1. lépés: hexadecimálisról binárissá alakítás
   $12 (18) = 00010010
2. lépés: a felső félbyte GCR kóddá alakítása
   0001xxxx = $1 (1) = 01011
3. lépés: az alsó félbyte GCR kóddá alakítása
   xxxx0010 = $2 (2) = 10010
4. lépés: GCR konkatenáció:
   01011+10010 = 0101110010

A konverziós technika következményei

1. Sohasem fordulhat elő, hogy két egymást követő ötbites GCR kód összes bitjén 1-es áll, azaz sosem keletkezhet GCR kódokból 10 egymást követő 1-es bit, amit a rendszer szinkronjelként használ.

2. A bináris GCR konverzió kizárja azt a lehetőséget, hogy az író/olvasó elektronika az adatbyte-ot összetévessze a szinkronjellel.

3. A tízbites GCR byte-ban egymás mellett sohasem fordulhat elő kettőnél több 0 tartalmú (alacsony) bit. (Erre a bitek olvasás közbeni visszabillentésének pontossága miatt volt szükség.)

Az előbbiek alapján világos a valódi adatok és a szinkronjelek megkülönböztetése. A szinkronjel 10 vagy annál több egymást követő 1-es bit. A valódi adatok közül a $FF (%11111111) tartalmazza a leghosszabb 1-es értékű bitsorozatot. Normál írásmódban a rendszer a $FF bináris megfelelőjét GCR kóddá, azaz az 1010110101 bitsorozattá alakítja. Szinkronmódban a $FF-et GCR végrehajtása nélkül írja a lemezre. Mivel a $FF csak nyolc egymást követő 1-es bitet tartalmaz, és a szinkronjeleknek legkevesebb tíz 1-es bitből kell állniuk, a Commodore öt nyolcbites $FF-et, azaz negyven 1-es értékű bitet ír fel folytatólagosan egymás mögé. Ez tökéletes garancia arra, hogy a 1541-es írás/olvasás közben sose tévedhessen a szinkronjel és az adatblokk megkülönböztetésében.

A 1541-es DOS egyszerre négy nyolcbites adatbyte-ot konvertál négy tízbites GCR byte-tá. Abból, hogy a RAM rekeszei csak nyolcbitesek, egyenesen következik, hogy a tízbites GCR kód nem fér el egy tárcímen. Ha egymás mögött négy tízbites GCR kódot tárolunk, a keletkezett 40 bitet a gép már képes nyolcas egységekben kezelni. A probléma most már csak az, hogy hogyan bontsa fel a 40 GCR bitet öt nyolcbites byte-ra. A következőkben lépésről lépésre követjük a DOS konverziós tevékenységét.

1. lépés: adott négy nyolcbites adatbyte
   $08 $10 $00 $12
2. lépés: hexadecimális bináris átalakítás

$08	$10	$00	$12
00001000	00010000	00000000	00010010

3. lépés: bináris GCR átalakítás
a) négy nyolcbites adatbyte
0001000 00010000 00000000 00010010
b) a felső és alsó félbyte-ok GCR megfelelői
01010 01001 01011 01010 01010 01010 01011 10010
c) négy tízbites GCR byte
4. lépés: a négy tízbites GCR kód átalakítása négy nyolcbites GCR kóddá
a) a négy tízbites GCR kód láncolása
0101001001010110101001010010100101110010
b) felosztás öt nyolcbites byte-ra
01010010 01010110 10100101 00101001 01110010
5. lépés: bináris–hexadecimális átalakítás
01010010 01010110 10100101 00101001 01110010
$52 $56 $A5 $29 $72
6. lépés: a négy nyolcbites adatbyte rögzítése öt nyolcbites
GCR byte-ként;
a $08 $10 $00 $12 tárolt megfelelői:
$52 $56 $75 $29 $72

Tehát a DOS a lemezen az adatbyte-okat négyesével tárolja, öt nyolcbites GCR byte alakjában. A 1541-es olvasás közben az eljárás fordítottját végzi el; az öt GCR byte-ot négy nyolcbites adatbyte-ként olvassa vissza. A lépések pontosan követik a fentieket, fordított sorrendben.

A szektorban tárolt byte-ok számát a következő képlettel kapjuk:
a nyolcbites GCR byte-ok száma = (nyolcbites adatbyte-ok száma/4)*5

Hasonlóan, a 260 nyolcbites adatbyte-nak a lemezen 325 nyolcbites GCR byte felel meg. Ehhez még hozzá kell adni az öt nyolcbites szinkronjel byte-ot és a fejblokk gap-et, ami további nyolc byte. A gap tartalmát ($55) a rendszer nem hozza GCR alakra, feladata csak az adatblokk és a fejblokk elválasztása. A teljes szektor 353, és nem 256 byte-ból áll:

15. táblázat

	Adatbyte-ok száma	GCR byte-ok száma
Szinkronjel ($FF)	5*	5
Fejblokk	8	10
Fej-gap ($55)	8*	8
Szinkronjel ($FF)	5*	5
Adatblokk	260	325

* tárolás bináris - GCR konverzió nélkül

Az egy szektorhoz tartozó byte-ok összegéből szándékosan kihagytuk a szektorok közötti gap-et. Ennek az a magyarázata, hogy a DOS a formázás befejeztével erre a gap-re soha nem hivatkozik. Formázás közben az FDC (Floppy Disk Controller) a sávot 10 240 nyolcbites $FF átfedő karakterrel törli. Törlés után felír rá 2400 nyolcbites $FF-et (%11111111-ot), majd mögé 2400 nyolcbites $55-öt (%01010101-et). A végső cél az, hogy a sávot világosan elkülöníthető byte-minta vegye körül. Ekkor az FDC megszámolja, hogy összesen hány szinkronjel ($FF), és hány egyéb (nem szinkronjel, $55) jel került a sávra. Ebből levonja az összes byte-ok számát, amelyek az adott zónában elhelyezkedő szektorhoz szükségesek. A maradékot elosztja a szektorok számával, és így megkapja a szektorokat elhatároló gap-ek méretét. A gap-ek mérete nemcsak sávonként, a kör kerületétől és a szektorok számától függően, de meghajtóként, a motor sebességétől függően is változó. Ha a számítások eredményeként a gap méretére négy byte-nál kevesebbre adódik, a formázás sikertelen lesz. A gap-ek közelítő méretét a 16. táblázat tartalmazza:

16. táblázat

Zóna	Sáv	Szektorok száma	Elhárító gap-ek mérete
1	1–17	21	4–7
2	18–24	19	9–12
3	25–30	18	5–8
4	31–35	17	4–8

Meg kell jegyeznünk, hogy ezek az értékek nem vonatkoznak a sáv legmagasabb sorszámú szektorára. Ezek között és a 0-s szektor között a gap általában jóval nagyobb (100 byte is lehet). A fejblokkot a rendszer a formázást követően soha nem írja felül. Ezzel szemben a szektor adatblokkját – beleértve a szinkronjelet is – mindig újraírja, valahányszor új adatok kerülnek a szektorba. A DOS a fejblokkot felírás közben átugorja.

4.1.2.2 Az ellenőrző összeg

Szemben a kazettás egységgel – amely az adatokat (pl. a programot) kétszer egymás után rögzíti a szalagon –, a lemezegység minden adatot csak egyszer ír fel a lemezre. Az esetleges hibák kiszűrésének eszköze a lemez esetében az ellenőrző összeg. A DOS az ellenőrző összeg alapján dönti el, hogy a fejblokk vagy az adatblokk tartalma hibás vagy sem. Az ellenőrző összeget a byte-ok között végzett kizáró VAGY (EOR) művelet eredménye adja. A művelet igazságtáblája:

0 EOR 0 = 0
0 EOR 1 = 1
1 EOR 0 = 1
1 EOR 1 = 0

A fejblokk ellenőrző összege a szektor és a sáv sorszáma és az ID alsó, ill. felső byte-ja között végzett EOR művelet eredménye (ez a négy byte különbözteti meg egymástól a lemez szektorait). Az adatblokk ellenőrző összegét pedig a 256 nyolcbites adatbyte között végzett EOR művelet eredménye adja.

Az adatblokk (amint azt már említettük) a következő szektor és sáv címétől és 254 adatbyte-ból áll. Megjegyzendő, hogy a DOS az ellenőrző összeget a GCR átalakítás előtt képezi.

A következő példa a fejblokk ellenőrző összegének kiszámítását szemlélteti. (Az algoritmus adatblokk esetén ugyanaz, de jóval hosszabb.)

1. lépés: inicializálás; EOR $00 (0) és a szektorok sorszáma között

$00 = 00000000
a szektor sorszáma $00 = 00000000
00000000

2. lépés: EOR a sáv sorszámával

00000000
a sáv sorszáma ($12) = 00010010
00010010

3. lépés: EOR az ID alsó byte-jával

00010010
az ID alsó byte-ja ($58) = 01011000
01001010

4. lépés: EOR az ID felső byte-jával

01001010
az ID felső byte-ja ($5A) = 01011010
00010000

5. lépés: bináris–hexadecimális átalakítás:
00010000 = $10 (=16)

A $00, $12, $58 és $5A byte-ok ellenőrző összege tehát $10 (16), ami történetesen a 1541 TEST/DEMO lemez 18-as sáv 0-s szektora fejblokkjának felel meg. (Az előzőekben a GCR konverziót ugyanezen fejblokk első négy byte-ján ($08, $10, $00, $12) mutattuk be.)

4.1.3 A 1541-es DOS részletes áttekintése

A 1541-es lemezegység saját 6502-es mikroprocesszorral, 2 Kbyte RAM-mal, I/O chipekkel rendelkezik, operációs rendszere a DOS, amely a 15,8 Kbyte méretű ROM-ban található. A következőkben a RAM, a ROM és az I/O chipek elrendezését szemléltetjük:

2 Kbyte RAM	input/output chipek
$0000	jobsor, konstansok, mutatók és munkaterület $1800 6522-es VIA CHIP I/O a gép felé
$0100	munkaterületek és a túlcsordulási puffer $180F
$0200	parancspuffer és munkaterület $1C00 6522-es VIA CHIP I/O a gép felé
$0300	adatpuffer (#0) $1C0F
$0400	adatpuffer (#1)
$0500	adatpuffer (#2)
$0600	adatpuffer (#3) $C100 DOS a 15,8 Kbyte ROM-ban kommunikáció és file-kezelés
$0700	BAM-puffer $F259 lemezvezérlő rutinok
$0800	$FFFF

4.1.4 A 6502-es processzor

A 1541-es a Commodore lemezegységek sorozatának utolsó tagja. A korábbi 2040-es, 4040-es, 8050-es egységekbe három mikroprocesszort építettek be: egy 6502-est a számítógép és a lemezegység közötti kommunikáció lebonyolítására, egy 6504-est lemezvezérlőként, végül egy 6532-est a nyolcbites karakterek tízbites GCR kódokká alakítására. A 1541-es mindezen feladatokat egyetlen 6502-es processzorral látja el. Kétféle üzemmódban dolgozik; betölti egyfelől az Interface Processor (IP), másfelől a Floppy Disk Controller (FDC) szerepét.

Működés közben minden 10 milliszekundumban kapcsol át FDC üzemmódra. Az átkapcsolást a 6522-es timer által generált megszakítás (IRQ) vezérli. A központi IRQ vezérlőrutin ellenőrzi, hogy az adott időpillanatban volt-e megszakítás. Ha a megszakítási jel az FDC üzemmódot engedélyezi, a 6502-es mindaddig ebben az üzemmódban dolgozik, amíg az aktuális műveleteket be nem fejezte. Ha a megszakítás nem engedélyezett, az írás/olvasás félbeszakadhat.

4.1.4.1 A legfontosabb IP rutinok

Ebben a fejezetben összefoglaltuk a legfontosabb IP rutinokat, és közöljük a belépési pontjukat.

a) Inicializálás

A lemezegység bekapcsolásakor a RESET vonal alacsony. Ez a 6502-es processzort egy indirekt JMP ugrásra készteti a $FFFC vektoron keresztül, a $EAA0-ás címen kezdődő inicializáló eljárásra. Az inicializálás főbb lépései:

$EAA0	A nulláslap tesztelése
$EAC9	A ROM-ok ellenőrző összegének tesztelése
$EAF0	A RAM maradék részének tesztelése
$EB22	Az I/O chipek inicializálása
$EB4B	A puffertáblázatok létrehozása
$EB87	A puffermutatók beállítása
$EBC2	Ugrás az FDC inicializálására
$EBDA	A soros busz inicializálása

b) Az IP főbb várakozó ciklusai

Valahányszor az egység inaktív, a 6502-es IP üzemmódban van, és végrehajtja a $EBE7-től $EC9D-ig terjedő rutint. A 4. ábra a rutin vázlatos folyamatábrája:

A várakozó parancs kapcsolója ($0255) igen A várakozás parancs elemzése és végrehajtása. ISR PARSXQ ($C146)
nem
A »figyelem« kapcsoló ($0255) magas? igen A »figyelem« kérelem kiszolgálása. ISR ATMSRV ($E85B)
nem
Van nyitott file? igen A »lemezegység aktív« LED bekapcsolása
nem
A hibakapcsoló magas? igen
nem
Ugrás a ciklus kezdetére.

4. ábra

c) Számítógép–lemezegység kommunikáció

Ezek a rutinok bonyolítják a számítógép és a lemezegység közötti információcserét a soros buszon keresztül. A rutinok a ROM $E853-tól $EA6E-ig terjedő területén vannak. Az alábbiakban a legfontosabb belépési pontokat soroljuk fel.

17. táblázat

Belépési pont	Rutin	Feladat
$E853	ATNIRQ	IRQ-t generál, ha a gép soros buszon az ATN vonalat magasra állítja. Az IRQ kezelőtől elágazás ide, a »figyelem« kapcsoló beállításához
$E8553	ATNSRV	A soros busz ATN jelének kiszolgálása
$E909	TALK	Adatok küldése a soros buszon keresztül
$E9C9	ACPTR	Egy byte fogadása a soros buszról
$EA2E	LISTEN	A soros buszon érkező jel átvétele

d) A lemezparancsok végrehajtása

Amikor a számítógép parancsot (mint pl. NEW, VERIFY, SCRATCH) küld a lemezegységhez, a parancs átmenetileg a parancspufferbe ($0200–$0229) kerül, és a »parancsra vár« kapcsoló ($0255) magas lesz. Amikor a 6502-es legközelebb az IP várakozó ciklushoz ér ($EBE7–$EC9D), észreveszi, hogy a kapcsoló állapota magas. Elugrik (JSR) a PARSXQ rutinra ($C146) a parancsot elemezni és végrehajtani. Az elemző azonosítja a parancs érvényességét a táblázatban ($FE89–94), majd ellenőrzi a szintaktikáját. Ha a parancs helyes, a megfelelő ROM rutinra ugrik. A 18. táblázatban a lemezparancsokat és belépési pontjaikat soroljuk fel.

18. táblázat

Belépési pont	Parancs	Feladat
$ED84	**V** VALIDATE	Új BAM létrehozása a tartalomjegyzékben
$D005	**I** INITIALIZE	A BAM inicializálása a lemez alapján
$C8C1	**D** DUPLICATE	A lemez másolása (nem 1541-esen)
$CAF8	**M** MEMORY-OP	Egy memóriaművelet (M-R, M-W, M-E) végrehajtása
$CC1B	**B** BLOCK-OP	Egy blokkművelet (B-P, B-A, B-F stb.) végrehajtása

18. táblázat

Belépési pont	Parancs	Feladat
$CD57	**U** USER JMP	User rutinok (U0, U1, U2 stb.) végrehajtása
$E207	**P** POSITION	Pozicionálás relatív file-ban
$E7A3	**&** UTIL LDR	Egy rutin betöltése a RAM-ba, és végrehajtása
$C8F0	**C** COPY	Egy file másolása (egylemezes, csak a 1541-esen)
$CA88	**R** RENAME	Egy file átnevezése a tartalomjegyzékben
$C823	**S** SCRATCH	Egy file törlése a tartalomjegyzékből
$EE0D	**N** NEW	A lemez formázása

Ha a parancs végrehajtása hibátlan, a rutin egy JMP utasítással fejeződik be, az ENDCMD ($C194) címre. Ha hiba lépett fel, a hiba kódja az akkumulátorba töltődik, és a vezérlés a hibakereső rutinra ($E645) adódik.

e) File-kezelés

A file-kezelés az interface processzor legfontosabb feladata. Éppen ezért számos ROM rutin foglalkozik közvetlenül vagy közvetve file-kezeléssel, a tartalomjegyzékkel és a BAM-mal. A 19. táblázatban felsorolunk néhányat a legfontosabb belépési pontok közül:

19. táblázat

Belépési pont	Rutin	Feladat
$C5AC	SRCHST	Érvényes vagy törölt bejegyzés keresése a tartalomjegyzékben
$CBB4	OPNBLK	A közvetlen elérésű puffer megnyitása
$CE0E	FNDREL	Egy rekord keresése a relatív file-ban
$D156	RDBYT	Egy byte beolvasása a file-ból. A következő szektor betöltése, ha szükséges
$D19D	WRTBYT	Egy byte felírása a file-ba. Egy szektor felírása, ha megtelt
$D50E	SETJOB	A read vagy write job beállítása az FDC-nek
$D6E4	ADDFIL	Egy file bejegyzése a tartalomjegyzékbe

19. táblázat folytatása

Belépési pont	Rutin	Feladat
$D7B4	OPEN	Egy csatorna megnyitása olvasásra, írásra, betöltése vagy mentése
$DAC0	CLOSE	Az adott csatornához rendelt file lezárása
$DBA5	CLSDIR	A tartalomjegyzék-bejegyzés lezárása write file-nál
$DC46	OPNRCH	Egy csatorna megnyitása dupla pufferrel végzett olvasáshoz
$DCDA	OPNWCH	Egy csatorna megnyitása dupla pufferrel végzett íráshoz
$DFD0	NXTREC	A következő rekord összeállítása relatív file-hoz
$E31F	ADDREL	Új szektor hozzáadása a relatív file-hoz
$E44E	NEWSS	Relatív file bővítése egy új mellékszektorral
$E4FC	ERRTAB	IP üzemmódban a hibaüzenetek táblázata
$E645	CMDERZ	IP üzemmódban a hibakezelő
$EA6E	PEZRO	Hibajelzés a LED villogtatásával
$EAA8	DSKINT	IP inicializálás lemezoldalról
$EC9E	STDIR	A tartalomjegyzék konvertálása pszeudoprogrammá, és betöltése
$EF5C	WFREE	Egy adott szektor felszabadítása a BAM-ban
$EF90	WUSED	Egy adott szektor lefoglalása a BAM-ban
$F11E	NXTTS	A legközelebbi szabad szektor azonosítása a BAM-ban

4.1.4.2 A legfontosabb FDC rutinok

Az FDC rutinok alkalmazásának az a legnehezebb pontja, amíg megtaláljuk a részletes ROM térképen azt a részt, amely pontosan azt a feladatot végzi el, amit szeretnénk. Éppen ezért foglaltuk össze röviden az FDC legfontosabb belépési pontjait.

a) Inicializálás

Amikor a lemezegységet bekapcsoljuk, a reset vonal alacsonyra ugrik. Ez a 6502-es processzort a RESET végrehajtására szólítja fel. A rutin a $FFFC szektoron keresztül közvetett JUMP-pal ráugrik a $EAA0-ás címen kezdődő inicializáló eljárásra. Az inicializáló eljárás részeként, az FDC I/O chipjeit és változóit a CNTINT ($F259-AF) rutin hozza alaphelyzetbe.

b) Az FDC várakozó ciklusa

A 6522-es timer minden 10 milliszekundumban megszakítást (IRQ-t) generál, és a 6502-es elkezdi végrehajtani a fő FDC ciklust ($F2B0). A ciklus vázlatos folyamatábrája (5. ábra):

```
Van job a jobsorban?  nem  JMP az END-re
igen
JMP ($D0) job?  igen  A JMP job ($F370) végrehajtása
nem
Be kell kapcsolni a motort?  igen  Motor ON és JMP az END-re
nem
Az egység felgyorsult?  nem  JMP az END-re
igen
A fej továbblépett?  nem  JMP az END-re
igen
Ez az igazi sáv?  igen  A job elvégzése
nem
A sáv beállítása lépéshez
END  ($F99C)
Változott az írásvédelem?  igen  Az állapotkapcsoló módosítása ($1C)
nem
A fej két sáv között van?  igen  A fej mozgatása JMP DOSTEP ($FA2E)
nem
Ki kell kapcsolni a motort?  igen  Motor OFF
nem
A fej léptető módban van?  igen
nem
RTS az IRQ rutinra
```

5. ábra

A ciklus végén, amikor a művelet kész, a timer-megszakítás újra engedélyezett, és a 6502-es kilép az FDC üzemmódból.

c) Az FDC főbb belépési pontjai

A 6502-es processzor FDC üzemmódban olyan rutinokat hajt végre, amelyek közvetlenül vezérlik a lemezegység műveleteit. Ide tartozik a lemezmotor ki/bekapcsolása, a léptetőmotor vezérlése (azaz a fej mozgatása sávról sávra), az üres lemez formázása, az egyes szektorok azonosítása, adatok felírása és olvasása, az információ továbbítása a nyolcbites normál alakról a tízbites GCR kódra, és megfordítva. A 6502-es a műveleteket az IP processzor által létrehozott jobsor alapján rendezi. Az FDC főbb belépési pontjait a 20. táblázatban foglaltuk össze.

20. táblázat

Belépés	Rutin	Feladat
$F259	CNTINT	A fontosabb változók és az I/O chipek inicializálása
$F2B0	LCC	A fő FDC várakozó ciklus (IRQ minden 10 milliszekundumban)
$F367	EXE	A job végrehajtása
$F37C	BMP	A fej „kiugratása" (bump) a #1 sávra (léptetés a 45. sávon kívülre)
$F3B1	SEAK	A fej azonosítása egy sávon
$F4CA	READ	A kiválasztott szektor adatblokkjának beolvasása
$F56E	WRIGHT	Adatblokk felírása a kiválasztott szektorba
$F6D0	PUT4GB	Négy adatbyte konvertálása öt GCR byte-tá
$F78F	BINGCR	Az adatpuffer tartalmának konvertálása GCR írásképpé
$F7E6	GET4GB	Öt GCR byte konvertálása négy adatbyte-tá
$F8E0	GCRBIN	Az adatblokk GCR képének konvertálása normál adatokká
$F934	CONHDR	A fej konvertálása GCR kereső képpé
$F99C	END	A ciklus vége
$FAC7	FORMT	Üres lemez formázása

d) Egy meghatározott szektor adatblokkjának beolvasása

Mielőtt az olvasás jobkódja ($80) bekerül a jobsorba, az IP beteszi a kiválasztott sáv–szektor számát a fejtáblázatba. A táblázat a következőképpen épül fel:

A jobsor elhelyezkedése	A pufferek kiosztása # cím	Sáv # cím	Szektor # cím
$0000	0 $0300–FF	$0006	$0007
$0001	1 $0400–FF	$0008	$0009
$0002	2 $0500–FF	$000A	$000B
$0003	3 $0600–FF	$000C	$000D
$0004	4 $0700–FF	$000E	$000F
$0005	5 nem RAM	$0010	$0011

A sáv–szektor számának elhelyezése után az IP elhelyezi az olvasás jobkódját a jobsorban úgy, hogy pozíciója megfeleljen az adat adatpufferbeli pozíciójának. Amikor a 6502-es legközelebb FDC módba kerül, megtalálja a jobkérelmet. Ha szükséges, bekapcsolja a motort, vár, amíg a motor felgyorsul, és a fejet a megfelelő sávon mozgatja. Ezután végrehajtja a read rutint a következők szerint:

$F4D1 A szektor megkeresése
$F4D4 Az adatok beolvasása: az első 256 byte az adatpufferbe, a maradék a túlcsordulási pufferbe kerül
$F4ED A GCR kódok konvertálása normál alakra
$F4F0 Az adatblokk ID ellenőrzése
$F4FB Az ellenőrző összeg vizsgálata
$F505 Kilépés, az olvasás rendben

e) Adatok felírása az adott szektorba

Mielőtt az írás jobkódja ($90) bekerül a jobsorba, az IP a sáv és szektor számát elhelyezi a fejtáblázatban. A táblázat felépítése azonos az előzővel. A write rutint előkészítő tevékenységek azonosak a read rutinnál leírtakkal. A write rutin szerkezete:

$F575 Az ellenőrző összeg kiszámítása
$F57A Az írásvédelem ellenőrzése
$F586 A puffer konvertálása GCR alakra
$F589 A szektor megkeresése
$F58C Várakozás a fej gap-re
$F594 Átkapcsolás írásra és öt szinkronjel ($FF) kiírása
$F5B1 A túlcsordulási puffer kiírása
$F5BF Az adatpuffer kiírása

$F5CC Átkapcsolás olvasásra
$F5D9 A GCR kód konvertálása nyolcbites alakra
$F5DC A jobkód módosítása: VERIFY
$F5E6 Ellenőrzés (verify)

f) Üres lemez formázása

Az IP formázó rutin ($C8C6) beír egy JMP $FAC7 utasítást a $0600-as címre, majd beteszi az EXECUTION jobkódját ($E0) a jobsorba ($0003). A következő várakozó állásban az FDC megtalálja a jobkódot, és végrehajt egy ugrást a formázórutinra, amelynek felépítése a következő:

$FAC7 Ellenőrzés: ez az első belépés? Ha nem, elágazás a $FAF5-re
$FACB Bump a #1-es sávra (hangjel!)
$FAE3 A hibák számának inicializálása, kilépés
$FAF5 A sávszám ellenőrzése
$FB00 Az írásvédelem ellenőrzése
$FB0C A sáv törlése szinkronjelekkel
$FB0F A sáv felének teleírása szinkronjellel, a másik fele nem szinkronjellel
$FB35 A szinkron- és nem szinkronjelek mérése
$FB7D Az időtartamok összehasonlítása, és a tail-gap-ek számának meghatározása
$FC36 A fejek képének kialakítása
$FC86 Az üres adatblokk kialakítása
$FC8C A fej konvertálása GCR alakra
$FC9E Az adatblokk konvertálása GCR alakra
$FCAA Egy szektor kiírása
$FD24 Átkapcsolás olvasásra és ellenőrzésre (verify)
$FD83 Ha minden szektor rendben, tovább a következő sávra
$FD96 Ha minden sáv kész, kilépés

Az első sorban mindaddig várunk, míg a V kapcsoló (BYTE–READY) nem lesz magas, utalva arra, hogy a byte az olvasásra készen áll. Ezután a V kapcsolót „kézzel" töröljük, majd a byte-ot az akkumulátorba töltjük.

Egy byte felírásának programja:

```
    STA  $1C01
C1  BVC  C1
    CLV
```

A byte-ot tároljuk, majd várakozunk arra, hogy a BYTE–READY vezeték a kapcsolót beállítsa, majd azt ismét töröljük.

A lemezegységet tájékoztatnunk kell arról, hogy a $1C01 címen tárolt adatot felírni (és nem beolvasni) szeretnénk.
Erre szolgál a következő program:

```
LDA  $1C0C   kontrollregiszter
AND  ##1F
ORA  #$C0
STA  $1C0C
LDA  #$FF    kimenetre
STA  $1C03   adatirány-regiszter; A port
```

Hasonlóan programozhatjuk az olvasási kérelmet:

```
LDA  $1C0C
ORA  #$E0
STA  $1C0C
LDA  #$00    bemenetre
STA  $1C03
```

Ezeket a rutinokat a DOS is tartalmazza. Aktivizálhatjuk a

```
JSR  $FE00
```

utasítással.

A lemezműveletek (írás/olvasás) közben gyakran kell a szinkronjelre várakozni. A lemezegység operációs rendszere erre is tartalmaz megfelelő rutint. Hívó utasítása:

```
JSR  $F556
```

A rutin kb. 20 mikroszekundumig várakozik a SYNC byte-ra. Ha ez idő alatt nem találja meg, hibaüzenetet küld. Ebben rejlik egyébként a hívásra alapozott várakozás hátránya. A programozónak ui. nincs lehetősége arra, hogy a hibát saját elképzelése szerint kezelje; a hibaüzenet kiírása után a vezérlés már nem kerül vissza a rutinhoz.

A megoldás ennek ellenére egyszerű. Ha a rutint beépítjük a saját programunkba, a negatív eredménnyel záruló lekérdezés után tetszőleges, saját rutinra elágazhatunk. Ha a SYNC byte beolvasása sikeres volt, a tényleges lekérdező rutin használata nélkül, a megfelelő bit leolvasásával is tájékozódhatunk a hiba felől.

```
C1   BIT  $1C00
     BMI  C1
```

Ennyit bevezetésként a lemezegység programozásáról. Végül röviden összefoglaljuk a VIA2 fontosabb regisztereit.

4.1.4.3 A VIA2 fontos regiszterei

$1C00: Drive Control Bus

Bit	Jelentés
0	A léptetőmotor vezérlése
1	A léptetőmotor vezérlése
2	A lemezegység motorja; 0 = kikapcsolva
3	LED; 0 = kikapcsolva
4	Írásvédelem; 0 = kikapcsolva
5	A bitsebesség beállítása
6	Ua., mint az 5. bit
7	A SYNC byte azonosítása olvasásnál; 0 = a SYNC byte megvan

$1C01: Az A port adatbeviteli regisztere – R/W fej adatbusz

$1C02: A B port adatirány-regisztere

$1C03: Az A port adatirány-regisztere

$1C04: 1-es timer (alsó byte)

$1C05: 1-es timer (felső byte)

$1C06: 1-es timer

Írás/olvasás közben a timer értéke változatlan. Ez a cím átmeneti tároló, a mindenkori érték megőrzése. Ha a $1C11-es cím hatodik bitje 1 (FREE-RUNNING-MODE), az érték minden nullára futásánál automatikusan a számlálóba kerül

$1C07: 1-es timer (ld. a $1C06 címnél leírtakat)

$1C08: 2-es timer (alsó byte)

$1C09: 2-es timer (felső byte)

$1C0A: Eltolási regiszter

$1C0B: Segédregiszter a vezérléshez

Bit	Jelentés
0	PA (Latch-Enable-Disable)
1	PB (0 = disable; 1 = enable)
2	Eltolást vezérlő bit
3	Eltolást vezérlő bit
4	Eltolást vezérlő bit
5	T2 timer-vezérlés (0 = időbeli megszakítás; 1 = várakozó számlálása a PB6 jele alapján)
6	T1 timer-vezérlés
7	T1 timer-vezérlés

$1C0C: perifériaregiszter (PCR)

Bit	Jelentés
0	CA1 megszakításvezérlés (0 = negatív él; 1 = pozitív él)
1	CA2 vezérlés
2	CA2 vezérlés

3	CA2	vezérlés
4	CB1	megszakításvezérlés (ld. a 0. bitet)
6	CB2	vezérlés
7	CB2	vezérlés

$1C0D: Interrupt Flag Register
Ez a regiszter jelzi az egyes bitek 1 értékére rendelt tevékenység eredményét

Bit	Jelentés
0	Aktív él a CA2-n
1	Aktív él a CA1-en
2	Nyolc eltolási pulzus az SR-ről ($1C0A)
3	Aktív él a CB2-n
4	Aktív él a CB1-en
5	A 2-es timer lefutott
6	Az 1-es timer lefutott
7	Magas, ha ebben a regiszterben a $1C0E címen egy bit magas

$1C0E: Interrupt Enable Register. A regiszter bitjei rendre megfelelnek a $1C0D regiszter bitjeinek. Ha valamelyik bit magas és a $1C0D bitje a megfelelő állapotot jelzi, a rendszer IRQ-t hajt végre

$1C0F: A port adatregisztere
Tartalma a $1C00 cím tükörképe, handshake üzemmód nélkül

4.2 Védelem hibás sávokkal, szektorokkal

A lemezek védelmi rendszerének alapjai általában a hibás szektorok. Ez egyben azt is jelenti, hogy a védelem létrehozója szándékosan ront el bizonyos szektorokat. A lemez eredetiségének vizsgálata legtöbbször abban áll, hogy egy rövid betöltő program beolvassa az elrontott sávot, és ellenőrzi a jelszót. A jelszó általában az FDC vagy IP által generált hibakód. Ha azt a betöltő program „helyesnek" találja, engedélyezi a tényleges program indítását. A lemezvédelem szerzőjének az az érdeke, hogy a betöltő programot a lehető legjobban elrejtse. Ha ui. a betöltő program védelmét valakinek sikerül feltörnie, a tényleges programhoz könnyűszerrel hozzáfér. Persze a dolog általában nem olyan egyszerű, mint ahogyan hangzik. A betöltőt többnyire olvashatatlan, a 6502-es használaton kívüli műveleti kódjaival írják meg, autostarttal látják el. Sokszor könnyebb az egész lemezt bitenként átolvasni, mint a betöltő titkosítását megfejteni.

Mielőtt áttekintenénk néhány sáv, ill. szektorrongáló módszert, az azonosított hibák felmerülésének sorrendjében röviden összefoglaljuk a DOS hibaüzeneteit.

4.2.1 A DOS hibaüzenetei

Olvasási hibák

22. táblázat

FDC jobkérelem	FDC hibakód	IP hibakód	Hibaüzenet
SEEK	$03 (3)	21	Nincs szinkronjel
SEEK	$02 (2)	20	A fejblokk nem azonosítható
SEEK	$09 (9)	27	Hibás a fejblokk ellenőrző összege
SEEK	$0B (11)	29	ID-hiba
READ	$02 (2)	20	A fejblokk nem azonosítható
READ	$04 (4)	22	Az adatblokk hiányzik
READ	$05 (5)	23	Hibás az adatblokk ellenőrző összege
READ	$01 (1)	0	OK

Írási hibák

23. táblázat

FDC jobkérelem	FDC hibakód	IP hibakód	Hibaüzenet
WRITE	–	73	DOS verzióeltérés
WRITE	$0B (11)	29	Hibás lemez ID
WRITE	$08 (8)	26	Írásvédelem bekapcsolva
WRITE	$07 (7)	25	Verify-hiba írás közben
VERIFY	$01 (1)	0	OK

21 READ ERROR (NO SYNC CHARACTER) Az FDC nem találja a szinkronjelet (tíz vagy annál több 1-es értékű bitből álló sorozatot) egy adott sávon az előírt 20 milliszekundumon belül (time out hiba).

20 READ ERROR (HEADER BLOCK NOT FOUND) Az FDC 90 egymás utáni kísérlet során sem találta meg a GCR fejblokkot ($52). Az FDC sávot keres és szinkronjelet talál. Ekkor beolvassa a következő byte-ot, és tartalmát ismét összehasonlítja $52-vel ($08 címen). Az összehasonlítás eredménye negatív, és a kísérletek számát jelző változó értéke eggyel csökken. Az FDC ismét szinkronjelre vár, és a kísérletet megismétli (összesen kilencvenszer).

27 READ ERROR (CHECKSUM ERROR IN HEADER BLOCK) Az FDC hibás fejblokkot talált a sávon. Ez a hiba a RAM-ba olvasás után derült ki; itt ui. a GCR byte-okat eredeti bináris alakra konvertálná. Ezután EOR műveletet végez a szektor és a sáv sorszáma, az ID alsó és felső byte-ja között. A kapott és a fejblokkban talált ellenőrző összeg között ismét EOR műveletet végez. Ha az eredmény nem nulla, a két összeg nem volt egyenlő, és így az FDC a $09 hibakódot generálja.

29 READ ERROR (DISK ID MISMATCH) A fejblokkban talált és a $0012, $0013 címeken tárolt (master copy) ID nem azonosak. Az utóbbit a rendszer a lemez inicializálása közben tölti be a nulláslap két megadott címére a lemez 18-as sávjáról. (Ugyanez történik egy adott sáv keresése közben is.)

20 READ ERROR (HEADER BLOCK NOT FOUND A fejblokk GCR képét a rendszer a szektor, a sáv sorszáma és a master lemez ID-je alapján hozza létre. Az FDC megkísérelte megtalálni a sávon belül azt a fejet, amely megegyezik a RAM-ban tárolt GCR kóddal. A hibaüzenet generálását kilencven sikertelen kísérlet előzte meg.

22 READ ERROR (DATA BLOCK NOT PRESENT) Egy adott sáv és szektor fejblokkját a rendszer átadja az előző vizsgálathoz. Az FDC megtalálta az adatblokk szinkronjelét és beolvasta a következő 325 GCR byte-ot a RAM-ba. A GCR byte-okat 260 nyolcbites bináris byte-tá alakítja. Dekódolás után az első byte tartalmát összehasonlítja a $0047-es címen tárolt adatblokk-azonosítóval, és hibát jelez, ha nem egyezik. A nulláslapon a jelzett cím tartalma alaphelyzeben $07.

23 READ ERROR (CHECKSUM ERROR IN DATA BLOCK) A 256 adatblokkból a rendszer ellenőrző összeget képez. A kapott összeg nem egyezik a lemezről beolvasott ellenőrző összeggel.

73 DOS MISMATCH (CBM DOS V2.6 1541) A rendszer olyan lemezre próbált írni, amelynek formátuma nem kompatibilis a rendszerrel. A DOS verziószáma a $0101-es címen nem $41. A tárcím tartalmát a rendszer általában a lemez inicializálása közben módosítja, a lemez 18-as sáv 0-s szektor első két byte-ja alapján.

26 WRITE PROTECT ON Írásra tettünk kísérletet, miközben az írásvédelmi kapcsoló be van kapcsolva. Távolítsuk el a lemezről az írásvédelmi tapaszt.

25 WRITE VERIFY ERROR Az éppen most felírt adatblokk tartalma visszaolvasás után nem egyezik meg a RAM-ban tárolt adatokkal. A hiba oka valószínűleg egy folt a lemezen. Az eredmény lehet egy lezáratlan file. Adjuk ki a validate parancsot, hogy a rendszer a BAM-ot újraszervezze.

4.2.2 Hibás blokkot készítő programok

20-as hiba egy szektoron

A program egy adott sáv, adott szektorában 20-as olvasási hibát generál. Az eljárás során először azonosítjuk a megelőző szektor fejét és adatblokkját, így meghatározhatjuk a következő szektor kezdetét. A blokkfej képét a RAM $24-től $2C-ig terjedő területén hozzuk létre. Ezek után átkapcsolunk írásmódra, és felírjuk a blokkfejet.

```
100 REM 20-AS HIBA EGY SZEKTORON
110 PRINT"■■■ 20 HIBA SAV-SZEKTOR-1541"
120 PRINT"■■■ KEREM A LEMEZT."
130 INPUT"SAV/SZEKTOR";T,S
140 IF T<1ORT>35THENEND
150 NS=20+2*(T>17)+(T>24)+(T>30)
160 IFS<0ORS>NSTHENEND
170 OPEN15,8,15
180 PRINT#15,"I0"
190 INPUT#15,EN$,EM$,ET$,ES$
200 IFEN$="00"GOTO250
210 PRINT"■■■"EN$","EM$","ET$","ES$
220 CLOSE15
230 END
240 REM SEEK
250 IFS=0THENS=NS:GOTO270
260 S=S-1
270 JB=176
280 GOSUB490
290 REM READ
300 JB=128
310 GOSUB490
320 I=0:PRINT"PROGRAM BEIRASA■"
330 READ HD$:IF VAL(HD$)=-1 THEN380
340 GOSUB 1110
350 PRINT#15,"M-W"CHR$(1)CHR$(5)CHR$(8)CHR$(D)
360 I=I+1
370 GOTO 330
380 REM VEGREHAJTAS
390 PRINT#15,"M-W"CHR$(2)CHR$(0)CHR$(1)CHR$(224)
400 PRINT#15,"M-R"CHR$(2)CHR$(0)
410 GET#15,E$
420 IFE$=""THENE$=CHR$(0)
430 E=ASC(E$)
440 IFE>127GOTO400
450 CLOSE15
460 PRINT"■■■■■KESZ !"
470 END
480 REM JOB QUEUE
490 TR=0
500 PRINT#15,"M-W"CHR$(8)CHR$(0)CHR$(4)CHR$(
501 T)CHR$(S)CHR$(T)CHR$(S)
510 PRINT#15,"M-W"CHR$(1)CHR$(0)CHR$(1)CHR$(JB)
520 TR=TR+1
530 PRINT#15,"M-R"CHR$(1)CHR$(0)
540 GET#15,E$
550 IFE$=""THENE$=CHR$(0)
```

```
560 E=ASC(E$)
570 IFTR=500GOTO600
580 IFE>127GOTO520
590 RETURN
600 CLOSE15
610 PRINT"[redacted]HIBAS.....!"
620 END
630 REM GEPIKODU PROGRAM
640 DATA 20,10,F5:REM       JSR   F510
650 DATA 20,56,F5:REM       JSR   F556
660 DATA A0,14    :REM       LDY  #14
670 DATA A5,19    :REM       LDA   19
680 DATA C9,12    :REM       CMP  #12
690 DATA 90,0C    :REM       BCC   051A
700 DATA 88       :REM       DEY
710 DATA 88       :REM       DEY
720 DATA C9,19    :REM       CMP  #19
730 DATA 90,06    :REM       BCC   051A
740 DATA 88       :REM       DEY
750 DATA C9,1F    :REM       CMP  #1F
760 DATA 90,01    :REM       BCC   051A
770 DATA 88       :REM       DEY
780 DATA E6,18    :REM       INC   18
790 DATA C5,18    :REM       CMP   18
800 DATA 90,06    :REM       BCC   0526
810 DATA F0,04    :REM       BEQ   0526
820 DATA A9,00    :REM       LDA  #00
830 DATA 85,19    :REM       STA   19
840 DATA A9,00    :REM       LDA  #00
850 DATA 45,16    :REM       EOR   16
860 DATA 45,17    :REM       EOR   17
870 DATA 45,18    :REM       EOR   18
880 DATA 45,19    :REM       EOR   19
890 DATA 85,1A    :REM       STA   1A
900 DATA 20,34,F9:REM       JSR   F934
910 DATA 20,56,F5:REM       JSR   F556
920 DATA A9,FF    :REM       LDA  #FF
930 DATA 8D,03,1C:REM       STA   1C03
940 DATA AD,0C,1C:REM       LDA   1C0C
950 DATA 29,1F    :REM       AND  #1F
960 DATA 09,C0    :REM       ORA  #C0
970 DATA 8D,0C,1C:REM       STA   1C0C
980 DATA A2,00    :REM       LDX  #00
990 DATA B5,24    :REM       LDA   24,X
1000 DATA 50,FE   :REM        BVC   054B
1010 DATA B8      :REM        CLV
1020 DATA 8D,01,1C:REM        STA   1C01
1030 DATA E8      :REM        INX
1040 DATA E0,08   :REM        CPX  #08
1050 DATA D0,F3   :REM        BNE   0549
1060 DATA 50,FE   :REM        BVC   0556
1070 DATA 20,00,FE:REM        JSR   FE00
1080 DATA A9,01   :REM        LDA  #01
1090 DATA 4C,69,F9:REM        JMP   F969
1100 DATA -1      :REM  PROGRAM VEGE
1110 REM HEXABOL DEC.-BE
1120 H$="0123456789ABCDEF"
1130 IF LEN(HD$)=0 THEN PRINT"HIBA":STOP
1140 IF LEN(HD$)<> 2 THEN HD$=LEFT$(HD$,2)
1150 H=0
```

```
1160 FOR ZZ=1 TO 16
1170 IF LEFT$(HD$,1)=MID$(H$,ZZ,1) THEN H=ZZ:GOTO 1190
1180 NEXT ZZ
1190 H=H-1
1200 L=0
1210 FOR ZZ=1 TO 16
1220 IF RIGHT$(HD$,1)=MID$(H$,ZZ,1) THEN L=ZZ:GOTO 1240
1230 NEXT ZZ
1240 L=L-1
1250 D=H*16+L
1260 PRINT1,HD$;"□"
1270 RETURN
READY.
```

21-es hiba a teljes sávon

A programot a $FAC7 címen kezdődő formátrutin egy részére alapoztuk. A sávot formázás előtt az FDC teleírja szinkronjelekkel ($FDA3).

```
100 REM          21 HIBA TELJES SAVON
110 PRINT"■■■■■■■■■■21 HIBA   -  1541-RE"
120 PRINT"■■■ KEREM A LEMEZT A MEGHAJTOBA"
130 INPUT"■■■ ♥AV=";T
140 IFT<1 OR T>35 THEN 130
150 OPEN15,8,15
160 PRINT#15,"I0"
170 INPUT#15,EN$,EM$,ET$,ES$
180 IFEN$="00"GOTO230
190 PRINT"■■■"EN$", "EM$","ET$", "ES$
200 CLOSE15
210 END
220 REM SEEK - JOB VEGREHAJTASA
230 JB=176
240 GOSUB380
250 I=0
255 PRINT"■ PROGRAM BETOLTES....■"
260 READ HD$:IF VAL(HD$)=-1 THEN 300
270 GOSUB 630
275 PRINT I,HD$;"□"
280 PRINT#15,"M-W"CHR$(I)CHR$(4)CHR$(1)CHR$(D)
290 I=I+1: GOTO 260
300 REM VEGREHAJTAS
310 PRINT"■■■ ■EGRONGALT■ SAV";T
320 JB=224 :REM EXECUTE
330 GOSUB380
340 PRINT"■■■ VEGE!"
350 CLOSE15
360 END
370 REM JOB QUEUE
380 TX=0
390 PRINT#15,"M-W"CHR$(8)CHR$(0)CHR$(2)CHR$(T)CHR$(0)
400 PRINT#15,"M-W"CHR$(1)CHR$(0)CHR$(1)CHR$(JB)
410 TX=TX+1
420 PRINT#15,"M-R"CHR$(1)CHR$(0)
430 GET#15,E$
440 IFE$=""THENE$=CHR$(0)
450 E=ASC(E$)
460 IFTX=500GOTO490
470 IFE>127GOTO410
480 RETURN
```

```
490 CLOSE15
500 PRINT"■■ ■IIBA■"
510 END
520 REM 21 ERROR
530 DATA 20,A3,FD:REM        JSR   FDA3    IRASRA KAPCSOLAS
540 DATA A9,55   :REM        LDA  #55      SYNC-BYTE
550 DATA 8D,01,1C:REM        STA   1C01
560 DATA A2,FF   :REM        LDX  #FF
570 DATA A0,30   :REM        LDY  #30
580 DATA 20,C9,FD:REM        JSR   FDC9    BYTE READY?
590 DATA 20,00,FE:REM        JSR   FE00    $55 FELIRASA
600 DATA A9,01   :REM        LDA  #01
610 DATA 4C,69,F9:REM        JMP   F969
620 DATA -1      :REM  PROGRAM VEGE
630 REM HEXABOL DEC.-BE
640 H$="0123456789ABCDEF"
650 IF LEN(HD$)=0 THEN PRINT"HIBA":STOP
660 IF LEN(HD$)<> 2 THEN HD$=LEFT$(HD$,2)
670 H=0
680 FOR ZZ=1 TO 16
690 IF LEFT$(HD$,1)=MID$(H$,ZZ,1) THEN H=ZZ:GOTO 710
700 NEXT ZZ
710 H=H-1
720 L=0
730 FOR ZZ=1 TO 16
740 IF RIGHT$(HD$,1)=MID$(H$,ZZ,1) THEN L=ZZ:GOTO 760
750 NEXT ZZ
760 L=L-1
770 D=H*16+L
780 RETURN
READY.
```

21-es hiba egy szektoron

A gépi kódú rutin megkeresi az előző szektort, majd azon belül az adatblokkot. A rutin 20-as hibát generál azon a szektoron, ahol áll (lásd az előző programot). Egyetlen mellékhatása ennek a módszernek az, ha két egymást követő szektort rongálunk meg, akkor mindkettőn 21-es hiba keletkezik. Tehát a rutin egyetlen szektoron nem tud 21-es hibát generálni, legalább két egymást követő szektort kell megrongálni. Figyeljünk arra is, hogy az alábbi rendben generáljuk a hibákat, először a 0-s szektort kell tönkretenni, majd rendre a 20-as, 19-es és 18-as szektorokat.

Szektor	**Hibakód**
0	21
1–17	OK
18–20	21

```
10 REM 21-ES HIBA ADOTT SAV/SZEKTORRA
120 PRINT"■■■ 21 HIBA SAV-SZEKTOR-1541"
130 PRINT"■■■ KEREM A LEMEZT."
140 INPUT"SAV/SZEKTOR";T,S
150 IF T<1ORT>35THENEND
160 NS=20+2*(T>17)+(T>24)+(T>30)
```

```
170 IFS<0ORS>NSTHENEND
200 OPEN15,8,15
210 PRINT#15,"I0"
220 INPUT#15,EN$,EM$,ET$,ES$
230 IFEN$="00"GOTO280
240 PRINT"■■"EN$","EM$","ET$","ES$
250 CLOSE15
260 END
270 REM SEEK
280 IFS=0THENS=NS:GOTO300
290 S=S-1
300 JB=176
310 GOSUB570
320 REM READ
330 JB=128
340 GOSUB570
350 REM READ DATA
355 I=0:PRINT"PROGRAMTOLTES....■"
360 READ HD$: IF VAL(HD$)=-1 THEN 460
370 GOSUB 5000
430 PRINT#15,"M-W"CHR$(I)CHR$(5)CHR$(1)CHR$(D)
440 I=I+1
450 GOTO 360
460 REM VEGREHAJTAS
470 PRINT#15,"M-W"CHR$(2)CHR$(0)CHR$(1)CHR$(224)
480 PRINT#15,"M-R"CHR$(2)CHR$(0)
490 GET#15,E$
500 IFE$=""THENE$=CHR$(0)
510 E=ASC(E$)
520 IFE>127GOTO480
530 CLOSE15
540 PRINT"■■■■■VESZ !"
550 END
560 REM JOB QUEUE
570 TR=0
580 PRINT#15,"M-W"CHR$(8)CHR$(0)CHR$(4)CHR$(
581 T)CHR$(S)CHR$(T)CHR$(S)
590 PRINT#15,"M-W"CHR$(1)CHR$(0)CHR$(1)CHR$(JB)
600 TR=TR+1
610 PRINT#15,"M-R"CHR$(1)CHR$(0)
620 GET#15,E$
630 IFE$=""THENE$=CHR$(0)
640 E=ASC(E$)
650 IFTR=500GOTO680
660 IFE>127GOTO600
670 IF E=1 THENRETURN
680 CLOSE15
690 PRINT"■■■■■■HIBAS.....!"
700 END
710 REM GEPIKOD SAV
1000 DATA 20,10,F5:REM      JSR   F510
1010 DATA 20,56,F5:REM      JSR   F556
1020 DATA A2,00   :REM      LDX  #00
1030 DATA 50,FE   :REM      BVC   0508
1040 DATA B8      :REM      CLV
1050 DATA CA      :REM      DEX
1060 DATA D0,FA   :REM      BNE   0508
1070 DATA A2,45   :REM      LDX  #45
1080 DATA 50,FE   :REM      BVC   0510
1090 DATA B8      :REM      CLV
```

```
1100 DATA CA        :REM        DEX
1110 DATA D0,FA     :REM        BNE   0510
1120 DATA A9,FF     :REM        LDA   #FF
1130 DATA 8D,03,1C:REM          STA   1C03
1140 DATA AD,0C,1C:REM          LDA   1C0C
1150 DATA 29,1F     :REM        AND   #1F
1160 DATA 09,C0     :REM        ORA   #C0
1170 DATA 8D,0C,1C:REM          STA   1C0C
1180 DATA A2,00     :REM        LDX   #00
1190 DATA A9,55     :REM        LDA   #55
1200 DATA 50,FE     :REM        BVC   0529
1210 DATA B8        :REM        CLV
1220 DATA 8D,01,1C:REM          STA   1C01
1230 DATA CA        :REM        DEX
1240 DATA D0,F7     :REM        BNE   0529
1250 DATA 50,FE     :REM        BVC   0532
1260 DATA 20,00,FE:REM          JSR   FE00
1270 DATA A9,01     :REM        LDA   #01
1280 DATA 4C,69,F9:REM          JMP   F969
1300 DATA -1          :REM    PROGRAM VEGE
5000 REM HEXABOL DEC.-BE
5010 H$="0123456789ABCDEF"
5020 IF LEN(HD$)=0 THEN PRINT"HIBA":STOP
5030 IF LEN(HD$)<> 2 THEN HD$=LEFT$(HD$,2)
5040 H=0
5050 FOR ZZ=1 TO 16
5060 IF LEFT$(HD$,1)=MID$(H$,ZZ,1) THEN H=ZZ:GOTO 5080
5070 NEXT ZZ
5080 H=H-1
5090 L=0
5100 FOR ZZ=1 TO 16
5110 IF RIGHT$(HD$,1)=MID$(H$,ZZ,1) THEN L=ZZ:GOTO 5130
5120 NEXT ZZ
5130 L=L-1
5140 D=H*16+L
5150 PRINT1,HD$;"␣"
5160 RETURN
READY.
```

23-as hiba egy szektorban

A programot a $F56E címen kezdődő rutinra* építjük, amelybe a 12. byte után lépünk be. Ezzel átugorjuk az írásvédelem ellenőrzését és az ellenőrző összeg kiszámítását. A rutin beolvassa a kijelölt szektor tartalmát a $400-tól $4FF-ig terjedő területre. Az indirekt puffermutatót is erre a területre címezzük. A $03A címen az ellenőrző összeget növeljük. Az 1-es puffer tartalmát konvertáljuk GCR alakra. Így a konvertálás során a 260 adatbyte-ból 325 nyolcbites GCR byte keletkezik. Ennek tárolásához egynél több pufferre van szükség. Az első 69 GCR byte-ot a $01BB–$01FF területre, egy túlcsordulási pufferbe helyezzük. A maradék 256 byte pedig a $400-tól $4FF-ig terjedő területen található. A 270-es sortól végigolvassuk a szektort, lemásoljuk a nyolcbyte-os header gap-et, és átkapcsolunk írásra. Felírunk lemezre öt $FF kódot (szinkronjelet), majd a túlcsordulási puffert és a tényleges puffer tartalmát. A rutin végrehajtása után betöltjük a 01-es hibakódot (OK).

* WRIGHT – a lektor megjegyzése

```
100 REM 23-AS HIBA EGY SZEKTORBAN
110 PRINT"■■■ 23 HIBA SAV-SZEKTORON"
120 PRINT"■■ KEREM A LEMEZT."
130 INPUT"SAV/SZEKTOR";T,S
140 IF T<1ORT>35THENEND
150 MS=20+2*(T>17)+(T>24)+(T>30)
160 IFS<0ORS>MSTHENEND
170 OPEN15,8,15
180 PRINT#15,"I0"
190 INPUT#15,EN$,EM$,ET$,ES$
200 IFEN$="00"GOTO250
210 PRINT"■■■"EN$","EM$","ET$",ES$
220 CLOSE15
230 END
240 REM SEEK
250 JB=176
260 GOSUB470
270 REM READ
280 JB=128
290 GOSUB470
300 I=0:PRINT"PROGRAMBETOLTES■"
310 READ HD$:IF VAL(HD$)=-1 THEN 360
320 GOSUB 1100
330 PRINT#15,"M-W"CHR$(I)CHR$(5)CHR$(8)CHR$(D)
340 I=I+1
350 GOTO 310
360 REM VEGREHAJTAS
370 PRINT#15,"M-W"CHR$(2)CHR$(0)CHR$(1)CHR$(224)
380 PRINT#15,"M-R"CHR$(2)CHR$(0)
390 GET#15,E$
400 IFE$=""THENE$=CHR$(0)
410 E=ASC(E$)
420 IFE>127GOTO380
430 CLOSE15
440 PRINT"■■■■■KESZ !"
450 END
460 REM JOB QUEUE
470 TR=0
480 PRINT#15,"M-W"CHR$(8)CHR$(0)CHR$(4)CHR$(
481 T)CHR$(S)CHR$(T)CHR$(S)
490 PRINT#15,"M-W"CHR$(1)CHR$(0)CHR$(1)CHR$(JB)
500 TR=TR+1
510 PRINT#15,"M-R"CHR$(1)CHR$(0)
520 GET#15,E$
530 IFE$=""THENE$=CHR$(0)
540 E=ASC(E$)
550 IFTR=500GOTO580
560 IFE>127GOTO500
570 RETURN
580 CLOSE15
590 PRINT"■■■■■■■HIBAS.....!"
600 END
610 REM GEPIKODU PROGRAM
620 DATA A9,04    :REM     LDA  #04
630 DATA 85,31    :REM     STA   31
640 DATA A5,3A    :REM     LDA   3A
650 DATA AA       :REM     TAX
660 DATA E8       :REM     INX
670 DATA 8A       :REM     TXA
680 DATA 85,3A    :REM     STA   3A
```

```
690 DATA 20,8F,F7:REM      JSR   F78F
700 DATA 20,10,F5:REM      JSR   F510
710 DATA A2,08    :REM      LDX  #08
720 DATA 50,FE    :REM      BVC   0513
730 DATA B8       :REM      CLV
740 DATA CA       :REM      DEX
750 DATA D0,FA    :REM      BNE   0513
760 DATA A9,FF    :REM      LDA  #FF
770 DATA 8D,03,1C:REM      STA   1C03
780 DATA AD,0C,1C:REM      LDA   1C0C
790 DATA 29,1F    :REM      AND  #1F
800 DATA 09,C0    :REM      ORA  #C0
810 DATA 8D,0C,1C:REM      STA   1C0C
820 DATA A9,FF    :REM      LDA  #FF
830 DATA A2,05    :REM      LDX  #05
840 DATA 8D,01,1C:REM      STA   1C01
850 DATA B8       :REM      CLV
860 DATA 50,FE    :REM      BVC   0530
870 DATA B8       :REM      CLV
880 DATA CA       :REM      DEX
890 DATA D0,FA    :REM      BNE   0530
900 DATA A0,BB    :REM      LDY  #BB
910 DATA B9,00,01:REM      LDA   0100,Y
920 DATA 50,FE    :REM      BVC   053B
930 DATA B8       :REM      CLV
940 DATA 8D,01,1C:REM      STA   1C01
950 DATA C8       :REM      INY
960 DATA D0,F4    :REM      BNE   0538
970 DATA B9,00,04:REM      LDA   0400,Y
980 DATA 50,FE    :REM      BVC   0547
990 DATA B8       :REM      CLV
1000 DATA 8D,01,1C:REM      STA   1C01
1010 DATA C8       :REM      INY
1020 DATA D0,F4    :REM      BNE   0544
1030 DATA 50,FE    :REM      BVC   0550
1040 DATA 20,00,FE:REM      JSR   FE00
1050 DATA A9,05    :REM      LDA  #05
1060 DATA 85,31    :REM      STA   31
1070 DATA A9,01    :REM      LDA  #01
1080 DATA 4C,69,F9:REM      JMP   F969
1090 DATA -1       :REM PROGRAM VEGE
1100 REM HEXABOL DEC.-BE
1110 H$="0123456789ABCDEF"
1120 IF LEN(HD$)=0 THEN PRINT"HIBA":STOP
1130 IF LEN(HD$)<> 2 THEN HD$=LEFT$(HD$,2)
1140 H=0
1150 FOR ZZ=1 TO 16
1160 IF LEFT$(HD$,1)=MID$(H$,ZZ,1) THEN H=ZZ:GOTO 1180
1170 NEXT ZZ
1180 H=H-1
1190 L=0
1200 FOR ZZ=1 TO 16
1210 IF RIGHT$(HD$,1)=MID$(H$,ZZ,1) THEN L=ZZ:GOTO 1230
1220 NEXT ZZ
1230 L=L-1
1240 D=H*16+L
1250 PRINT I,HD$;"D"
1260 RETURN
READY.
```

22-es hiba generálása

22-es hibát az előzőekhez hasonlóan (a job várakozási sorából indított programmal is) létrehozhatunk, de erre egy jóval egyszerűbb módszer is kínálkozik. Mint már korábban említettük, a $47-es címen található az adatblokk-azonosító, aminek tartalma alapesetben 07. Ha egy blokk felírása előtt ezt az értéket módosítjuk, úgy az adott szektoron egy 22-es típusú hibát hozhatunk létre.

```
100 PRINT"■"
110 INPUT"■■■■■  SV,SZ";SV,SZ
120 OPEN15,8,15
130 OPEN2,8,2,"#"
140 PRINT#15,"U1:"2;0;SV;SZ
150 PRINT#15,"B-P:"2;0
160 PRINT#15,"M-W";CHR$(71);CHR$(0);CHR$(1);CHR$(3)
170 A$=CHR$(65) :REM JELSZO=A
180 PRINT#2,A$
190 PRINT#15,"U2:"2;0;SV;SZ
200 PRINT#15,"M-W";CHR$(71);CHR$(0);CHR$(1);CHR$(7)
210 CLOSE2:CLOSE15
220 PRINT" A LEMEZ KESZEN VAN"
230 STOP
READY.
```

23-as hiba másolása

Először beolvassuk a megrongált adatblokkot, de az ellenőrző összeget sértetlenül hagyjuk. A szektor tartalmát a $400–$4FF-ig terjedő területen, az ellenőrző összeget a $3A címen tároljuk. Az összeget nem számoljuk újra és nem növeljük, hanem változatlanul felírjuk a másolati szektorba.

```
100 REM SAV-SZEKTOR MASOLAS
120 PRINT"■■■ 23 HIBA SAV-SZEKTOR MASOLASA  -1541"
130 PRINT"■■■ KEREM A MASOLANDO LEMEZT."
140 INPUT"SAV/SZEKTOR";T,S
150 IF T<1ORT>35THENEND
160 NS=20+2*(T>17)+(T>24)+(T>30)
170 IFS<0ORS>NSTHENEND
200 OPEN15,8,15
210 PRINT#15,"10"
220 INPUT#15,EN$,EM$,ET$,ES$
230 IFEN$="00"GOTO280
240 PRINT"■■■"EN$","EM$","ET$",ES$
250 CLOSE15
260 END
270 REM SEEK
280 JB=176
290 GOSUB650
300 REM READ
310 JB=128
320 GOSUB650
330 CLOSE15
340 PRINT"■■■ KEREM A CEL LEMEZT"
350 PRINT"■■■ ■RETURN■-RA TOVABB.."
360 GETC$:IFC$=""THEN360
370 IF C$<>CHR$(13)GOTO360
380 PRINT"OK...."
390 OPEN15,8,15
```

```
400 REM SEEK
410 JB=176
420 GOSUB650
430 I=0:PRINT"PROGRAMTOLTES■"
440 READ HD$:IF VAL(HD$)=-1 THEN 540
450 GOSUB 5000
510 PRINT#15,"M-W"CHR$(I)CHR$(5)CHR$(8)CHR$(D)
520 I=I+1
530 GOTO 440
540 REM VEGREHAJTAS
550 PRINT#15,"M-W"CHR$(2)CHR$(0)CHR$(1)CHR$(224)
560 PRINT#15,"M-R"CHR$(2)CHR$(0)
570 GET#15,E$
580 IFE$=""THENE$=CHR$(0)
590 E=ASC(E$)
600 IFE>127GOTO560
610 CLOSE15
620 PRINT"■■■■■KESZ !"
630 END
640 REM JOB QUEUE
650 TR=0
660 PRINT#15,"M-W"CHR$(8)CHR$(0)CHR$(4)CHR$(
661 T)CHR$(S)CHR$(T)CHR$(S)
670 PRINT#15,"M-W"CHR$(1)CHR$(0)CHR$(1)CHR$(JB)
680 TR=TR+1
690 PRINT#15,"M-R"CHR$(1)CHR$(0)
700 GET#15,E$
710 IFE$=""THENE$=CHR$(0)
720 E=ASC(E$)
730 IFTR=500GOTO760
740 IFE>127GOTO680
750 RETURN
760 CLOSE15
770 PRINT"■■■■■■HIBAS.....!"
780 END
790 REM GEPIKOD SAV
1000 DATA A9,04    :REM      LDA  #04
1010 DATA 85,31    :REM      STA   31
1020 DATA 20,8F,F7:REM       JSR   F78F
1030 DATA 20,10,F5:REM       JSR   F510
1040 DATA A2,08    :REM      LDX  #08
1050 DATA 50,FE    :REM      BVC   050C
1060 DATA B8       :REM      CLV
1070 DATA CA       :REM      DEX
1080 DATA D0,FA    :REM      BNE   050C
1090 DATA A9,FF    :REM      LDA  #FF
1100 DATA 8D,03,1C:REM       STA   1C03
1110 DATA AD,0C,1C:REM       LDA   1C0C
1120 DATA 29,1F    :REM      AND  #1F
1130 DATA 09,C0    :REM      ORA  #C0
1140 DATA 8D,0C,1C:REM       STA   1C0C
1150 DATA A9,FF    :REM      LDA  #FF
1160 DATA A2,05    :REM      LDX  #05
1170 DATA 8D,01,1C:REM       STA   1C01
1180 DATA B8       :REM      CLV
1190 DATA 50,FE    :REM      BVC   0529
1200 DATA B8       :REM      CLV
1210 DATA CA       :REM      DEX
1220 DATA D0,FA    :REM      BNE   0529
1230 DATA A0,BB    :REM      LDY  #BB
```

```
1240 DATA B9,00,01:REM      LDA   0100,Y
1250 DATA 50,FE   :REM      BVC   0534
1260 DATA B8      :REM      CLV
1270 DATA 8D,01,1C:REM      STA   1C01
1280 DATA C8      :REM      INY
1290 DATA D0,F4   :REM      BNE   0531
1300 DATA B9,00,04:REM.     LDA   0400,Y
1310 DATA 50,FE   :REM      BVC   0540
1320 DATA B8      :REM      CLV
1330 DATA 8D,01,1C:REM      STA   1C01
1340 DATA C8      :REM      INY
1350 DATA D0,F4   :REM      BNE   053D
1360 DATA 50,FE   :REM      BVC   0549
1370 DATA 20,00,FE:REM      JSR   FE00
1380 DATA A9,05   :REM      LDA  #05
1390 DATA 85,31   :REM      STA   31
1400 DATA A9,01   :REM      LDA  #01
1410 DATA 4C,69,F9:REM      JMP   F969
1420 DATA -1      :REM  PROGRAM VEGE
5000 REM HEXABOL DEC.-BE
5010 H$="0123456789ABCDEF"
5020 IF LEN(HD$)=0 THEN PRINT"HIBA":STOP
5030 IF LEN(HD$)<> 2 THEN HD$=LEFT$(HD$,2)
5040 H=0
5050 FOR ZZ=1 TO 16
5060 IF LEFT$(HD$,1)=MID$(H$,ZZ,1) THEN H=ZZ:GOTO 5080
5070 NEXT ZZ
5080 H=H-1
5090 L=0
5100 FOR ZZ=1 TO 16
5110 IF RIGHT$(HD$,1)=MID$(H$,ZZ,1) THEN L=ZZ:GOTO 5130
5120 NEXT ZZ
5130 L=L-1
5140 D=H*16+L
5150 PRINT I,HD$;"□"
5160 RETURN
READY.
```

22-es hiba másolása

A módszer azon alapul, hogy ha a sérült szektort megpróbáljuk beolvasni, a hibára utaló (az adatblokk kezdetét jelző) értéket, a rendszer tárolja az 56-os címen. Ezt kell kiolvasni (320-as sor) és visszaírni a 71-es címre (380-as sor). Így újból megkísérelve az olvasást, a szektor tartalmát már sértetlenül és hiánytalanul beolvashatjuk.

```
100 INPUT"SV,SZ";SV,SZ
110 GOSUB 190:REM E22(OLV)
120 PRINT"□"
130 PRINT"■■■■ LEMEZCSERE"
140 Z$=""
150 GETZ$:IFZ$=""THEN150
160 GOSUB 490:REM E22(IR)
170 PRINT" A MASOLAS MEGTORTENT"
180 STOP
190 REM SAV OLVASAS E22-RE
200 :
210 :
220 REM   SV=1:SZ=15
```

```
230 OPEN15,8,15
240 PRINT#15,"UJ"
250 POKE161,0
260 WAIT161,1
270 PRINT#15,"I"
280 OPEN2,8,2,"#"
290 PRINT#15,"U1:"2;0;SV;SZ
300 INPUT#15,A,B$,C$,D$
310 IF A<>22 THEN 410
320 PRINT#15,"M-R";CHR$(56);CHR$(0)
330 GET#15,H$
340 PRINT#15,"M-W";CHR$(71);CHR$(0);CHR$(1);H$
350 PRINT#15,"U1:"2;0;SV;SZ
360 INPUT#15,A,B$,C$,D$
370 IF A<>0 THEN 410
380 PRINT#15,"M-W";CHR$(71);CHR$(0);CHR$(1);CHR$(7)
390 CLOSE2:CLOSE15
400 RETURN
410 PRINTSV;" SAVON ES ";SZ;" SZEKTORBAN NEM"
420 PRINT"22-ES READ ERROR VAN"
430 PRINTA;B$
440 PRINT#15,"UJ"
450 POKE161,0
460 WAIT161,1
470 PRINT#15,"I"
480 CLOSE2:CLOSE15:RETURN
490 REM SV.SAV-RA SZ. SZEKTORBA 22 READ ERROR1 IR
500 :
510 :
520 REM    SV=1:SZ=15
530 :
540 :
550 OPEN15,8,15
560 PRINT#15,"I"
570 POKE161,0
580 POKE161,1
590 OPEN2,8,2,"#"
600 PRINT#15,"M-W";CHR$(71);CHR$(0);CHR$(1);H$
610 PRINT#2,A$
620 PRINT#15,"U2:"2;0;SV;SZ
630 PRINT#15,"M-W";CHR$(71);CHR$(0);CHR$(1);CHR$(7)
640 CLOSE2:CLOSE15
650 PRINT"A ";SV;"-RA ES ";SZ;
655 PRINT" SZEKTORBA A MASOLAS MEGTORTENT"
660 RETURN
READY.
```

Ennek a módszernek van még egy előnye is: ha tudjuk, hogy mit kell a 71-es címre betölteni, akkor a szektor tartalmát „zajtalanul" be tudjuk olvasni, így a lemez ellenőrzése szinte észrevétlenül megtörténhet. Hátránya viszont az, hogy a védelmet az előbbi programmal könnyen megismételhetik a másolati lemezen is.

4.3 Védelem a formázó rutin segítségével

A rendszer formázó rutinjának módosításával hathatós védelmet hozhatunk létre. Az új formátumot kialakíthatjuk egyetlen sávon is. A formázást kiterjeszthetjük a 36-tól 41-ig terjedő sávokra, és ha szükséges, az összes blokkfejparamétert megváltoztathatjuk.

4.3.1 Egy sáv formázása

Ennek lehetősége azon alapul, hogy a DOS a formázórutinját minden egyes sáv formázása előtt egy JMP $FAC7 utasításon keresztül hívja meg. Ezt az utasítást a $600-as címtől tárolja. Ezen a ponton beléphetünk a DOS-ba, és „kényszeríthetjük" a rutint arra, hogy egyetlen sávot formázzon meg. A feladat programozástechnikai szempontból egyetlen rövid kis program elkészítését igényli, amelyet az "E0" jobkóddal, a $03 jobtárból aktivizálunk.

```
100 REM            EGY SAV FORMAZASA
110 :
120 :
130 OPEN 1,8,15,"I"
140 READ HD$:IF VAL(HD$)=-1 THEN 190
150 GOSUB 430
160 PRINT#1,"M-W"CHR$(N)CHR$(6)CHR$(1)CHR$(D)
170 N=N+1
180 GOTO140
190 INPUT"SAV";T
200 IF T >35 OR T<1 THEN 190
210 PRINT#1,"M-W"CHR$(19)CHR$(6)CHR$(1)CHR$(T)
220 PRINT#1,"M-E"CHR$(18)CHR$(6)
230 PRINT#1,"M-R"CHR$(3)CHR$(0)CHR$(1)
240 GET#1,A$:A=ASC (A$+CHR$(0))
250 IF A>127 THEN 230
260 IFA =1 THEN PRINT"FORMAZAS RENDBEN":END
270 PRINT"FORMAZASI HIBA"
280 END
290 REM  PROGRAM **********************
300 DATA 4C,03,06 :REM 0600  JMP   0603
310 DATA A5,0C    :REM 0603  LDA   0C
320 DATA 85,51    :REM 0605  STA   51
330 DATA A9,0F    :REM 0607  LDA  #0F
340 DATA 8D,01,06 :REM 0609  STA   0601
350 DATA 4C,C7.FA :REM 060C  JMP   FAC7
360 DATA 4C,9E,FD :REM 060F  JMP   FD9E
370 DATA A9,01    :REM 0612  LDA  #01
380 DATA 85,0C    :REM 0614  STA   0C
390 DATA A9,E0    :REM 0616  LDA  #E0
400 DATA 85,03    :REM 0618  STA   03
410 DATA 60       :REM 061A  RTS
420 DATA -1       :REM VEGE
```

```
430 REM HEXABOL DEC.-BE
440 H$="0123456789ABCDEF"
450 IF LEN(HD$)=0 THEN PRINT"HIBA":STOP
460 IF LEN(HD$)<> 2 THEN HD$=LEFT$(HD$,2)
470 H=0
480 FOR ZZ=1 TO 16
490 IF LEFT$(HD$,1)=MID$(H$,ZZ,1) THEN H=ZZ:GOTO 510
500 NEXT ZZ
510 H=H-1
520 L=0
530 FOR ZZ=1 TO 16
540 IF RIGHT$(HD$,1)=MID$(H$,ZZ,1) THEN L=ZZ:GOTO 560
550 NEXT ZZ
560 L=L-1
570 D=H*16+L
580 RETURN
READY.
```

A gépi kódú programot a szokottnál részletesebben elemezzük, mert futás közben felülírja magát, s így másként fut a második hívás esetében. A könnyebb érthetőség kedvéért az utasítások előtt megadjuk tárbeli címüket is.

A BASIC program először inicializálja a lemezt, bitosítva ezzel, hogy a formázórutin helyes ID-t írjon a sávba. (Ha ui. az ID hibás, a sáv blokkjait a továbbiakban lehetetlen lenne külön erre a célra készített program nélkül beolvasni!) Ezután elhelyezzük a gépi kódú programot a lemeztárba a $600-as címtől kezdve (110–140).

Beolvassuk a formázó sáv számát a T változóba, és ezt egy M-W utasítás segítségével beleírjuk a gépi kódú program $613-as byte-jába. A következő programsor (180-as) egy M-E utasítás a $612 címen lévő rutinra. Ez a rutin betölti az akkumulátorba az előbb beírt sávszámot (tehát nem #01-et!), s tárolja a $0C címen, ahol a három puffer sávszámát tároljuk (lásd az egyes jobokhoz tartozó sáv–szektor cím, ill. tárterület-táblázatot). Ezután betöltjük a job végrehajtását kérő kódot, E0-t a 3-as pufferhez tartozó jobkód területre (03). Minthogy a Floppy Disk Controller (FDC) minden 10 ezredmásodpercben végignézi a jobkódterületeket, így „hamarosan" megtalálja az általunk beírt E0-s kódot is. Mivel ez egy programvégrehajtás kérése, ahol a program a 3 pufferben a $600-as címtől található, ezért az FDC ezt elkezdi végrehajtani. Az itt található első utasítás (ugrás a következő utasításra) csak első alkalommal felesleges! A következő két utasítás betölti a sor számát a $0C címről és tárolja a $51-es címen. A formázóprogram ui. erről a címről veszi veszi majd a következő formázandó sáv címét. Ha a cím tartalma $FF, akkor a formázás most kezdődik, ilyenkor a rendszer „kiakasztja" a fejet (végrehajt egy BUMP jobot). Ezután „beméri" a sávokat, hogy megállapítsa a szektorok közötti távolságot. (Ez a magyarázata egyébként annak, hogy a formázás viszonylag sok időt vesz igénybe.) Ekkor az egyes szektorokon létrehozza a blokkfejet, ami a sáv formázásával egyenértékű. Miután végighaladt az Összes szektoron, összehasonlítja az aktuális sávszámot 36-tal. Ha kisebb, a vezérlést arra a programra adja, amely a fejet egy kijelölt sávra pozicionálja, majd onnan a jobciklusra ugrik.

A jobciklusban ismét ellenőrzi, hogy minden job végrehajtása befejeződött-e, s ha a $03-as címen ismét a $E0 kódot találja, a formázást folytatja.

De térjünk vissza eredeti programunkhoz, ahol csak egy sávot szeretnénk megformázni! A következő két utasítás

```
LDA #0F
STA 0601
```

felülírja a #600-as címen kezdődő `JMP 0603` ugróutasítást, új tartalma `JMP 060F` lesz!

Ezek után végrehajtunk egy ugróutasítást a $FAC7 címre, ami a formázórutin eleje. Itt tehát megformázza a kijelölt sávot. Ezalatt természetesen önállóan fut a C 64-esen a BASIC program. Kiolvassa a $03-as cím tartalmát, ahová az FDC a művelet befejezése után egy hibaüzenet-kódot ír. Ha ez a kód nagyobb, mint 127 (a 7. bit 1-es), akkor itt még a jobkód található, a kért művelet még nem fejeződött be. Ellenkező esetben (7. bit 0) a művelet véget ért, az FDC által beírt kód ennek eredményét jelzi: ha ez az érték 1, akkor nem lépett fel hiba. A BASIC program tehát a 180 és 210 sorok között várakozik a művelet befejezésére. Itt van jelentősége a $600 címben lévő ugróutasításoknak, ha ui. a sáv formázása véget ért, a $612-es címen lévő rutin a $03 címre visszatölti a E0 jobkódot (180-as BASIC sor).

Az FDC ezt újból észreveszi, s végrehajtja a $600-as címtől elhelyezett programot. Itt azonban az ugróutasítás már más:

```
JMP 060F
```

(hiszen az előbb felülírtuk). Ezen a címen egy JMP FD9E utasítás van, amely a formázórutin befejezését végzi el. A rutin meghívására azért van szükség, hogy a formázórutint szabályosan fejezzük be. Ezzel a programmal 35-nél nagyobb sorszámú sávokat nem lehet formázni, ui. az már nem a program, hanem a lemezegység operációs rendszerének hatásköre.

A program csak a régebbi típusú lemezegységeken fut hibátlanul. Az újabb típusokhoz a 210-es sort módosítani kell:

```
210 IF A>127 THEN 190
```

A típusok közötti különbséget könnyen észrevehetjük, hiszen a lemez behelyezése után az újabb típusú készülékeken más formájú „kallantyúval" kell a lemezegység nyílását lezárni.

4.3.2 A 36-tól 41-ig sorszámozott sávok formázása

A lemezegység író/olvasó fejét minden néhézség nélkül a 35-ös sáv fölé mozgathatjuk. Ha azonban erre a területre adatokat is szándékozunk írni, néhány problémát előzetesen meg kell oldanunk. Az igaz, hogy a fejet a 35-ös sáv fölé mozgathatjuk, de az a táblázat, amelyben a DOS sávonkénti szektorok számát és a bitsebességre vonatkozó adatot keresi, a 36-nál nagyobb sorszámú sávokra semmilyen adatot nem tartalmaz. Az íráshoz/olvasáshoz szükséges paramétereket tehát ez esetben „saját kezűleg" kell a DOS-nak átadnunk.

A formázást ezzel a módszerrel is legfeljebb a 41-es sávig terjeszthetjük ki. A formázórutin tárgyalásakor említettük, hogy 35-nél nagyobb sávszám észlelésekor tevékenységét félbeszakítja. Éppen ezért a 36-nál nagyobb sorszámú sávokat csak a rutin ismételt újraindításával formázhatjuk meg, amire egyébként nincs szükség. Mindezen szempontok figyelembevételével készült a következő program, amelyet a lemeztár $0415-ös címéről kell elindítani (START).

```
100 REM            FORMAZAS 36-41 SAVRA
110 :
120 :
130 OPEN 1,8,15,"I"
140 READ HD$:IF VAL(HD$)=-1 THEN 190
150 GOSUB 550
160 SU=SU+D
170 PRINT#1,"M-W"CHR$(N)CHR$(4)CHR$(1)CHR$(D)
180 N=N+1:GOTO140
190 IF SU<> 5381 THEN PRINT"HIBAS A DATA":STOP
200 INPUT "SAV KEZDET >35";S
210 INPUT "SAV VEGE   <42";E
220 PRINT#1,"M-W"CHR$(25)CHR$(4)CHR$(1)CHR$(S)
230 PRINT#1,"M-W"CHR$(47)CHR$(4)CHR$(1)CHR$(E+1)
240 PRINT#1,"M-E"CHR$(21)CHR$(4)
250 PRINT#1,"M-R"CHR$(1)CHR$(0)CHR$(1)
260 GET#1,A$:A=ASC (A$+CHR$(0))
270 IF A>127 THEN250
280 IF A=1 THEN PRINT"FORMAZAS OK.":END
290 PRINT"FORMAZAS1 HIBA":END
300 DATA A9,11    :REM        LDA  #11
310 DATA 85,43    :REM        STA   43
320 DATA AD,00,1C:REM         LDA   1C00
330 DATA 29,9F    :REM        AND  #9F
340 DATA 09,00    :REM        ORA  #00
350 DATA 8D,00,1C:REM         STA   1C00
360 DATA A5,08    :REM        LDA   08
370 DATA 85,51    :REM        STA   51
380 DATA 4C,C7,FA:REM         JMP   FAC7
390 DATA 20,42,D0:REM START   JSR   D042
400 DATA A9,24    :REM        LDA  #24
410 DATA 85,37    :REM  C1    STA   37
420 DATA 85,08    :REM        STA   08
430 DATA A9,E0    :REM        LDA  #E0
440 DATA 85,01    :REM        STA   01
450 DATA A5,01    :REM  C2    LDA   01
```

```
460 DATA 30,FC    :REM         BMI   C2
470 DATA C9,02    :REM         CMP   #02
480 DATA B0,08    :REM         BCS   C3
490 DATA E6,37    :REM         INC   37
500 DATA A5,37    :REM         LDA   37
510 DATA C9,2A    :REM         CMP   #2A
520 DATA D0,E8    :REM         BNE   C1
530 DATA 60       :REM  C3     RTS
540 DATA -1       :REM  PROGRAM VEGE
550 REM HEXABOL DEC.-BE
560 H$="0123456789ABCDEF"
570 IF LEN(HD$)=0 THEN PRINT"HIBA":STOP
580 IF LEN(HD$)<> 2 THEN HD$=LEFT$(HD$,2)
590 H=0
600 FOR ZZ=1 TO 16
610 IF LEFT$(HD$,1)=MID$(H$,ZZ,1) THEN H=ZZ:GOTO 630
620 NEXT ZZ
630 H=H-1
640 L=0
650 FOR ZZ=1 TO 16
660 IF RIGHT$(HD$,1)=MID$(H$,ZZ,1) THEN L=ZZ:GOTO 680
670 NEXT ZZ
680 L=L-1
690 D=H*16+L
700 RETURN
READY.
```

Az utolsó sáv sorszáma nem lehet nagyobb 42-nél, ui. a pozicionálás ez esetben kiakasztaná az író/olvasó fejet, és „tönkretenné" a lemezegységet. A 42-es sáv a legtöbb jó minőségű lemezen még hibajelenség nélkül formázható.

A gépi kódú programot tehát a tár $415-ös címéről (START) indítjuk egy M-E utasítás segítségével (200-as sor). A $D042-es címen lévő szubrutin inicializálja a lemezt. Ezután betöltjük a kezdő sávszámot a $37 munkaterületre és a $08-as (1-es puffer) adatterületére. Beírjuk a jobkódot ($E0) a jobciklusba, így az FDC elindítja a $400-as címtől beírt programot. Itt betöltjük a sávonkénti szektorok számát (17-et) a $43-as címen lévő ún. szektorszámlálóba. Ezt a területet csak a formázórutin használja. Majd betöltjük a B adatkapu (periféria-regiszter) értékét, és beállítjuk a bitsebességet. Ezt követően elindíthatjuk a formázórutint.

A program futtatása utána a 36-tól 41-ig sorszámozott sávok készen állnak az adatok tárolására. Az adatok felírása azonban nyilvánvalóan nem olyan egyszerű, mint az alacsonyabb sorszámú sávok esetében. Amint már említettük, a DOS ezen „illegális" sávok adatait (a sebességet és a sávonkénti szektorok számát) nem tárolja. Éppen ezért a felírást sem az U1 (blokk olvasása), U2 (blokk írása) utasításokkal, sem a 80 (blokk olvasása) és 90 (blokk írása) jobkódokkal nem tudjuk programozni. Az egyetlen fennmaradó lehetőség az, hogy az E0 kódú jobbal aktivizálunk egy saját készítésű programot, amely a szükséges paramétereket a blokkot olvasó, ill. blokkot felíró rutinoknak átadja.

Erre az elvre alapozva bemutatunk egy programot, amelynek segítségével egy tetszőleges szöveget vagy jelszót írhatunk fel a lemezünkre, a 36. sávtól felfelé.

```
100 REM                  IR 36-41. SAVRA
110 :
120 :
130 OPEN2,8,2,"#2"
140 OPEN 1,8,15,"I"
150 READ HD$:IF VAL(HD$)=-1 THEN 200
160 GOSUB 580
170 SU=SU+D
180 PRINT#1,"M-W"CHR$(N)CHR$(6)CHR$(1)CHR$(D)
190 N=N+1:GOTO150
200 IF SU<> 3517 THEN PRINT"HIBAS DATA":STOP
205 PRINT"[redacted]  IRAS A 36-41. SAVOKRA"
210 INPUT "SAV";T
220 INPUT "SZEKTOR";S
230 INPUT "SZOVEG";TX$
240 FOR K=1 TO LEN(TX$)
250 K1$=MID$(TX$,K,1)
260 PRINT#1,"M-W"CHR$(K-1)CHR$(5)CHR$(1)K1$
270 PRINT#1,"M-W"CHR$(22)CHR$(6)CHR$(1)CHR$(T)
280 NEXT K
290 PRINT#1,"M-W"CHR$(26)CHR$(6)CHR$(1)CHR$(S)
300 PRINT#1,"M-E"CHR$(21)CHR$(6)
310 FOR N=1 TO 500:NEXT
320 PRINT#1,"M-R"CHR$(3)CHR$(0)CHR$(1)
330 GET#1,A$:A=ASC (A$+CHR$(0))
340 IF A>127 THEN300
350 CLOSE2
360 CLOSE1
370 IF A=1 THEN PRINT"OK":END
380 PRINT"HIBA":END
390 DATA AD,00,1C:REM          LDA   1C00
400 DATA 29,9F    :REM          AND  #9F
410 DATA 09,00    :REM          ORA  #00
420 DATA 8D,00,1C:REM          STA   1C00
430 DATA A9,11    :REM          LDA  #11
440 DATA 85,43    :REM          STA   43
450 DATA A9,05    :REM          LDA  #05
460 DATA 85,31    :REM          STA   31
470 DATA 4C,75,F5:REM          JMP   F575
480 DATA A9,24    :REM  START  LDA  #24
490 DATA 85,0C    :REM          STA   0C
500 DATA A9,00    :REM          LDA  #00
510 DATA 85,0D    :REM          STA   0D
520 DATA A9,E0    :REM          LDA  #E0
530 DATA 85,03    :REM          STA   03
540 DATA A5,03    :REM   C1     LDA   03
550 DATA 30,FC    :REM          BMI   C1
560 DATA 60       :REM          RTS
570 DATA -1       :REM  PROGRAM VEGE
580 REM HEXABOL DEC.-BE
590 H$="0123456789ABCDEF"
600 IF LEN(HD$)=0 THEN PRINT"HIBA":STOP
610 IF LEN(HD$)<> 2 THEN HD$=LEFT$(HD$,2)
620 H=0
630 FOR ZZ=1 TO 16
640 IF LEFT$(HD$,1)=MID$(H$,ZZ,1) THEN H=ZZ:GOTO 660
650 NEXT ZZ
660 H=H-1
670 L=0
680 FOR ZZ=1 TO 16
```

```
690 IF RIGHT$(HD$,1)=MID$(H$,ZZ,1) THEN L=ZZ:GOTO 710
700 NEXT ZZ
710 L=L-1
720 D=H*16+L
730 RETURN
READY.
```

A BASIC programrész beolvassa a sáv–szektor értékeit, valamint a jelszónak szánt azonosító szöveget. A szöveget beírja a lemeztár $500-as címétől lévő RAM területére; a sáv és szektor értékeivel pedig felülírja a programban konstansként tárolt értékeket. A gépi kódú programot a 300-as sorban lévő M-E utasítással indítjuk el (START címen). Itt a sáv, szektor és jobkód értékeket betöltjük a 3 puffer megfelelő területére, s megvárjuk, míg a $600-as címre beírt program lefut. Ez a programrészlet végzi el a bitsebesség meghatározását, illetve a sávonkénti szektorok számának megadását. A felírást a $F575 címen lévő rutin segítségével végezzük el, amely a 2-es pufferben tárolt szöveget a lemezre írja.

Az így felírt szöveget a következő kis programmal olvashatjuk vissza:

```
100 REM              OLVAS 36-41. SAVON
110 :
120 :
130 OPEN2,8,2,"#2"
140 OPEN 1,8,15,"I"
150 READ HD$:IF VAL(HD$)=-1 THEN 210
160 PRINTHD$
170 GOSUB 580
180 SU=SU+D
190 PRINT#1,"M-W"CHR$(N)CHR$(6)CHR$(1)CHR$(D)
200 N=N+1:GOTO150
210 IF SU<> 3608 THEN PRINT"HIBAS DATA":STOP
215 PRINT"■ OLVASAS 36-41. SAVOKON"
220 INPUT "SAV     ";T
230 INPUT "SZEKTOR";S
240 PRINT#1,"M-W"CHR$(22)CHR$(6)CHR$(1)CHR$(T)
250 PRINT#1,"M-W"CHR$(26)CHR$(6)CHR$(1)CHR$(S)
260 PRINT#1,"M-E"CHR$(21)CHR$(6)
270 FOR N=1 TO 500:NEXT
280 PRINT#1,"M-R"CHR$(3)CHR$(0)CHR$(1)
290 GET#1,A$:A=ASC (A$+CHR$(0))
300 IF A>127 THEN260
310 IF A<>1 THEN PRINT"HIBA":GOTO 370
320 FOR I=1 TO 10
330 PRINT#1,"M-R"CHR$(I-1)CHR$(5)CHR$(1)
340 GET#1,SZ$
350 PRINTSZ$;
360 NEXT
370 CLOSE2:CLOSE1
380 END
390 DATA AD,00,1C:REM          LDA   1C00
400 DATA 29,9F    :REM          AND  #9F
410 DATA 09,00    :REM          ORA  #00
420 DATA 8D,00,1C:REM          STA   1C00
430 DATA A9,11    :REM          LDA  #11
440 DATA 85,43    :REM          STA   43
```

```
450 DATA A9,05    :REM          LDA  #05
460 DATA 85,31    :REM          STA   31
470 DATA 4C,D1,F4:REM           JMP   F4D1
480 DATA A9,24    :REM  START   LDA  #24
490 DATA 85,0C    :REM          STA   0C
500 DATA A9,00    :REM          LDA  #00
510 DATA 85,0D    :REM          STA   0D
520 DATA A9,E0    :REM          LDA  #E0
530 DATA 85,03    :REM          STA   03
540 DATA A5,03    :REM    C1    LDA   03
550 DATA 30,FC    :REM          BMI   C1
560 DATA 60       :REM          RTS
570 DATA -1       :REM PROGRAM VEGE
580 REM HEXABOL DEC.-BE
590 H$="0123456789ABCDEF"
600 IF LEN(HD$)=0 THEN PRINT"HIBA":STOP
610 IF LEN(HD$)<> 2 THEN HD$=LEFT$(HD$,2)
620 H=0
630 FOR ZZ=1 TO 16
640 IF LEFT$(HD$,1)=MID$(H$,ZZ,1) THEN H=ZZ:GOTO 660
650 NEXT ZZ
660 H=H-1
670 L=0
680 FOR ZZ=1 TO 16
690 IF RIGHT$(HD$,1)=MID$(H$,ZZ,1) THEN L=ZZ:GOTO 710
700 NEXT ZZ
710 L=L-1
720 D=H*16+L
730 RETURN
READY.
```

A program a beolvasott szöveget ismét a 2-es pufferbe teszi be, ahonnan már a megfelelő M-R utasítások segítségével összerakhatjuk.

4.3.3 Kettőzött sávok

A most következő védelmi módszer sokkal inkább ötletessége, semmint bonyolultsága révén szembetűnő. Meglehetősen nehéz visszafejteni, ugyanis semmilyen lemezhibát nem jelez. A védelem lényege a formázórutin módosításában rejlik. Az ötlet nagyon egyszerű; az alsó sávokat megduplázzuk. Ez szó szerint azt jelenti, hogy a lemezen a megszokott 1-es, 2-es, 3-as sáv stb. helyett a sorrend 1-es sáv, 2-es sáv, 1-es sáv, 2-es sáv, 3-as sáv, 4-es sáv stb. lesz. Érdekes módon az 1-es és 2-es sávok duplán szerepelnek.

Ahhoz, hogy a feladatot meg tudjuk oldani, nézzük meg, hogyan keresi meg a DOS a lemezen az egyes sávokat. Először megvizsgálja, hogy az adott pillanatban hol áll az író/olvasó fej. Azután kiszámítja az aktuális hely és a keresett sáv közötti távolságot, és a fejet ennyi sávval mozgatja el. Ebből az következik, hogy a szokásos lépéseket követve a DOS sohasem éri el a legális 1-es sáv alatt elhelyezett illegális 1-es sávot.

Pontosan ugyanez vonatkozik a másolóprogramokra. Az illegális 1-es és 2-es sávok az író/olvasó fej számára „láthatatlanok". Akkor sincs probléma, ha a

lemezműveletek közben a fej kiakad, hiszen arra a sávra tér vissza, ahol előzőleg tartózkodott. Felismerve, hogy a kelleténél néhány sávval beljebb csúszott, visszatér az eredeti helyzetbe.

Az ötlet megvalósítása is nagyon egyszerű. Formázás előtt az író/olvasó fejet meghatározott számú sávval befelé kell mozgatni. A DOS anélkül, hogy erről tudomást venne, a formázást az általa 1-esnek tekintett sávon (valójában a szokottnál néhány sávval beljebb) kezdi. A következő BASIC program beolvassa a billentyűzetről a megkettőzendő sávok számát, és a lemezt a leírtak szerint megformázza.

```
100 REM                    AZONOS SAVOK
110 :
120 :
130 OPEN 1,8,15,"I"
140 READ HD$:IF VAL(HD$)=-1 THEN 200
150 GOSUB 770
160 SU=SU+D
170 PRINT#1,"M-W"CHR$(N)CHR$(4)CHR$(1)CHR$(D)
180 PRINTN,HD$
190 N=N+1:GOTO140
200 IF SU<> 9284 THEN PRINT"HIBAS DATA":STOP
210 INPUT"A DUPLIKALANDO SAVOK SZAMA<8 :";AN
220 INPUT"LEMEZ NEVE ";N$:N$=N$+","
230 FORN=0 TOLEN(N$)-1
240 PRINT#1,"M-W"CHR$(128+N)CHR$(4)CHR$(1)
241 MID$(N$,N+1,1):NEXT
250 PRINT#1,"M-W"CHR$(9)CHR$(4)CHR$(1)CHR$(AN*2)
260 PRINT#1,"M-E"CHR$(56)CHR$(4)
270 PRINT#1,"M-R"CHR$(1)CHR$(0)CHR$(1)
280 GET#1,A$:A=ASC(A$+CHR$(0))
290 IF A>127 THEN 270
300 IF A=1 THEN PRINT"FORMAT OK.":END
310 PRINT"FORMAT HIBA":END
320 END
330 DATA 4C,03,04:REM      JMP   0403
340 DATA A9,17   :REM      LDA  #17
350 DATA 8D,01,04:REM      STA   0401
360 DATA A9,04   :REM      LDA  #04
370 DATA 85,37   :REM      STA   37
380 DATA 20,1A,04:REM      JSR   041A
390 DATA C6,37   :REM      DEC   37
400 DATA D0,F9   :REM      BNE   040C
410 DATA A9,01   :REM      LDA  #01
420 DATA 85,51   :REM      STA   51
430 DATA 4C,C7,FA:REM      JMP   FAC7
440 DATA AE,00,1C:REM      LDX   1C00
450 DATA E8      :REM      INX
460 DATA 8A      :REM      TXA
470 DATA 29,03   :REM      AND  #03
480 DATA 85,4B   :REM      STA   4B
490 DATA AD,00,1C:REM      LDA   1C00
500 DATA 29,FC   :REM      AND  #FC
510 DATA 05,4B   :REM      ORA   4B
520 DATA 8D,00,1C:REM      STA   1C00
530 DATA A2,00   :REM      LDX  #00
540 DATA A0,05   :REM      LDY  #05
```

```
550 DATA CA        :REM      DEX
560 DATA D0,FD     :REM      BNE   0431
570 DATA 88        :REM      DEY
580 DATA D0,FA     :REM      BNE   0431
590 DATA 60        :REM      RTS
600 DATA A9,12     :REM      LDA   #12
610 DATA 85,08     :REM      STA   08
620 DATA A9,E0     :REM      LDA   #E0
630 DATA 85,01     :REM      STA   01
640 DATA A5,01     :REM      LDA   01
650 DATA 30,FC     :REM      BMI   0440
660 DATA C9,02     :REM      CMP   #02
670 DATA B0,11     :REM      BCS   0459
680 DATA A2,10     :REM      LDX   #10
690 DATA 8E,74,02:REM        STX   0274
700 DATA BD,80,04:REM        LDA   0480,X
710 DATA 9D,00,02:REM        STA   0200,X
720 DATA CA        :REM      DEX
730 DATA 10,F7     :REM      BPL   044D
740 DATA 4C,40,EE:REM        JMP   EE40
750 DATA 60        :REM      RTS
760 DATA -1        :REM  PROGRAM VEGE
770 REM HEXABOL DEC.-BE
780 H$="0123456789ABCDEF"
790 IF LEN(HD$)=0 THEN PRINT"HIBA":STOP
800 IF LEN(HD$)<> 2 THEN HD$=LEFT$(HD$,2)
810 H=0
820 FOR ZZ=1 TO 16
830 IF LEFT$(HD$,1)=MID$(H$,ZZ,1) THEN H=ZZ:GOTO 850
840 NEXT ZZ
850 H=H-1
860 L=0
870 FOR ZZ=1 TO 16
880 IF RIGHT$(HD$,1)=MID$(H$,ZZ,1) THEN L=ZZ:GOTO 900
890 NEXT ZZ
900 L=L-1
910 D=H*16+L
920 RETURN
READY.
```

Ha nem akarjuk, hogy az új formátum szembetűnő legyen, kettőnél több sávot nem célszerű megduplázni. A futtatás csak akkor lesz eredményes, ha a szokásos módon előkészített, hibátlanul formázott lemezt használunk. A program a lemez eredeti azonosítóját (az ID-t) nem változtatja meg. A sávok számának és a lemez nevének bekérése után a gépi kódú programot beírja a lemeztárba, és ott elindítja (START címkétől).

A következő program segítségével írhatunk az elrejtett 1-es sávra.

```
100 REM        IR A DUPLIKALT 1-ES SAVRA
110 :
120 :
130 OPEN2,8,2,"#2"
140 OPEN 1,8,15,"I"
150 READ HD$:IF VAL(HD$)=-1THEN190
160 GOSUB 650
170 SU=SU+D:PRINT#1,"M-W"CHR$(N)CHR$(6)CHR$(1)CHR$(D)
180 N=N+1:GOTO150
```

```
190 IF SU<>7053THEN PRINT"HIBAS DATA":STOP
200 INPUT "JELSZO";A$
210 FOR I=1 TO LEN(A$)
220 PRINTI
230 PRINT#1,"M-W"CHR$(I-1)CHR$(5)CHR$(1)MID$(A$,I,1)
240 NEXT I
250 PRINT#1,"M-E"CHR$(39)CHR$(6)
260 REM   PRINT#1,"U2";2;0;1;0
270 CLOSE1
280 END
290 DATA 20,09,06:REM       JSR   0609 A FEJ ELTOLASA
300 DATA 20,09,06:REM       JSR   0609
310 DATA 4C,9E,FD:REM       JMP   FD9E FORMAT END
320 DATA AE,00,1C:REM 0609  LDX   1C00 FDC B PORT
330 DATA CA       :REM       DEX
340 DATA 8A       :REM       TXA
350 DATA 29,03    :REM       AND  #03   AZ ELSO 2 BIT
360 DATA 85,4B    :REM       STA   4B   TAROLASA
370 DATA AD,00,1C:REM       LDA   1C00 MOZGATAS
380 DATA 29,FC    :REM       AND  #FC
390 DATA 05,4B    :REM       ORA   4B
400 DATA 8D,00,1C:REM       STA   1C00
410 DATA A2,00    :REM       LDX  #00
420 DATA A0,05    :REM       LDY  #05
430 DATA CA       :REM 0620  DEX
440 DATA D0,FD    :REM       BNE   0620
450 DATA 88       :REM       DEY
460 DATA D0,FA    :REM       BNE   0620
470 DATA 60       :REM       RTS
480 DATA 20,42,D0:REM KEZD  JSR   D042 LEMEZ INICIALIZALASA
490 DATA A9,01    :REM       LDA  #01   SAV SZAMA
500 DATA 85,0C    :REM       STA   0C
510 DATA A9,E0    :REM       LDA  #E0   EXECUTE
520 DATA 85,03    :REM       STA   03
530 DATA A5,03    :REM 0632  LDA   03   VARAKOZAS
540 DATA 30,FC    :REM       BMI   0632 A MUVELET VEGERE
550 DATA A0,01    :REM       LDA  #01   A DUPLIKALT SAV SZAMA
560 DATA 85,0A    :REM       STA   0A
570 DATA A9,00    :REM       LDA  #00   A SZEKTOR SZAMA
580 DATA 85,0B    :REM       STA   0B
590 DATA A9,90    :REM       LDA  #90   WRITE
600 DATA 85,02    :REM       STA   02
610 DATA A5,02    :REM 0642  LDA   02   VARAKOZAS AZ
620 DATA 30,FC    :REM       BMI   0642 IRAS VEGERE
630 DATA 60       :REM       RTS
640 DATA -1       :REM  PROGRAM VEGE
650 REM HEXABOL DEC.-BE
660 H$="0123456789ABCDEF"
670 IF LEN(HD$)=0 THEN PRINT"HIBA":STOP
680 IF LEN(HD$)<> 2 THEN HD$=LEFT$(HD$,2)
690 H=0
700 FOR ZZ=1 TO 16
710 IF LEFT$(HD$,1)=MID$(H$,ZZ,1) THEN H=ZZ:GOTO 730
720 NEXT ZZ
730 H=H-1
740 L=0
750 FOR ZZ=1 TO 16
760 IF RIGHT$(HD$,1)=MID$(H$,ZZ,1) THEN L=ZZ:GOTO 780
770 NEXT ZZ
780 L=L-1
```

```
790 D=H*16+L
800 RETURN
READY.
```

Mivel az egész eljárás célja a lemez másolás elleni védelme, tehát a védelem lekérdezéséről is gondoskodnunk kell. Az U1 utasítás természetesen most nem alkalmazható az elrejtett 1-es sáv tartalmának beolvasására.

A feladatot ismét saját készítésű lemezprogrammal oldhatjuk meg. Ez a program az író/olvasó fejet az eredeti 1-es sávra állítja, majd egy rutin segítségével egy sávval kijjebb tolja, hogy ott megkeresse az elrejtett 1-es sávot. Eközben a DOS-szal „el kell hitetnie", hogy a fej a valódi 1-es sávon áll. Valahányszor egy elrejtett sávot keresünk, a fejet először mindig az azonos sorszámú sávra kell beállítani. Ha ui. a fej az állítólagos 1-es sávon áll, a DOS észreveszi, hogy nem ez az első megformázott sáv, és tovább mozgatja a fejet lefelé, amíg az elrejtett sávot meg nem találja, és azt tekinti valódinak. Mindezek után a rejtett blokkok olvasását nyugodtan programozhatjuk az U1 utasítással. A következő gépi kódú program az elmondottak szerint működik. Kezdőcíme a lemeztárban $0627.

```
100 REM          OLVAS A DUPLIKALT 1-ES SAVROL
110 :
120 :
130 OPEN 1,8,15,"I"
140 READ HD$:IFVAL(HD$)=-1THEN180
150 GOSUB 640
160 SU=SU+D:PRINT#1,"M-W"CHR$(N)CHR$(6)CHR$(1)CHR$(D)
170 N=N+1:GOTO140
180 IF SU<>7037THEN PRINT"H1BAS DATA":STOP
190 PRINT#1,"M-E"CHR$(39)CHR$(6)
200 FOR I=1 TO 10
210 PRINT#1,"M-R"CHR$(I-1)CHR$(5)
220 GET#1,C$
230 K$=K$+C$
240 NEXT
250 PRINTK$
260 CLOSE1
270 END
280 DATA 20,09,06:REM      JSR   0609 A FEJ ELTOLASA
290 DATA 20,09,06:REM      JSR   0609
300 DATA 4C,9E,FD:REM      JMP   FD9E FORMAT END
310 DATA AE,00,1C:REM 0609 LDX   1C00 FDC B PORT
320 DATA CA       :REM      DEX
330 DATA 8A       :REM      TXA
340 DATA 29,03    :REM      AND  #03   AZ ELSO 2 BIT
350 DATA 85,4B    :REM      STA   4B   TAROLASA
360 DATA AD,00,1C:REM      LDA   1C00 MOZGATAS
370 DATA 29,FC    :REM      AND  #FC
380 DATA 05,4B    :REM      ORA   4B
390 DATA 8D,00,1C:REM      STA   1C00
400 DATA A2,00    :REM      LDX  #00
410 DATA A0,05    :REM      LDY  #05
420 DATA CA       :REM 0620 DEX
430 DATA D0,FD    :REM      BNE   0620
440 DATA 88       :REM      DEY
```

```
450 DATA D0,FA    :REM       BNE   0620
460 DATA 60       :REM       RTS
470 DATA 20,42,D0:REM KEZD   JSR   D042 LEMEZ INICIALIZALASA
480 DATA A9,01    :REM       LDA  #01   SAV SZAMA
490 DATA 85,0C    :REM       STA   0C
500 DATA A9,E0    :REM       LDA  #E0   EXECUTE
510 DATA 85,03    :REM       STA   03
520 DATA A5,03    :REM 0632  LDA   03   VARAKOZAS
530 DATA 30,FC    :REM       BMI   0632 A MUVELET VEGERE
540 DATA A9,01    :REM       LDA  #01   A DUPLIKALT SAV SZAMA
550 DATA 85,0A    :REM       STA   0A
560 DATA A9,00    :REM       LDA  #00   A SZEKTOR SZAMA
570 DATA 85,0B    :REM       STA   0B
580 DATA A9,80    :REM       LDA  #80   READ
590 DATA 85,02    :REM       STA   02
600 DATA A5,02    :REM 0642  LDA   02   VARAKOZAS AZ
610 DATA 30,FC    :REM       BMI   0642 OLVASAS VEGERE
620 DATA 60       :REM       RTS
630 DATA -1       :REM  PROGRAM VEGE
640 REM HEXABOL DEC.-BE
650 H$="0123456789ABCDEF"
660 IF LEN(HD$)=0 THEN PRINT"HIBA":STOP
670 IF LEN(HD$)<> 2 THEN HD$=LEFT$(HD$,2)
680 H=0
690 FOR ZZ=1 TO 16
700 IF LEFT$(HD$,1)=MID$(H$,ZZ,1) THEN H=ZZ:GOTO 720
710 NEXT ZZ
720 H=H-1
730 L=0
740 FOR ZZ=1 TO 16
750 IF RIGHT$(HD$,1)=MID$(H$,ZZ,1) THEN L=ZZ:GOTO 770
760 NEXT ZZ
770 L=L-1
780 D=H*16+L
790 RETURN
READY.
```

Az elvet és megvalósítását a lista alapján még egyszer áttekinthetjük. A program beolvassa és a 2-es pufferben tárolja az elrejtett 1-es sáv tartalmát.

Amint már említettük, a program futtatása után az elrejtett sávok blokkjai a szokott módon, a blokkolvasó és -író utasításokkal elérhetőek. Ha a védelem lekérdezése után a feldolgozást az eredeti lemezformátum szerint szeretnénk folytatni, elég, ha a lemezt inicializáljuk, vagy beolvasunk egy nem elrejtett sávot.

Ha valaki SPEED-DOS operációs rendszerrel dolgozik, akkor a lemez formázásához kapcsolja ki azt, ui. a formázórutin nem hajtja végre a BUMP jobot a formázás előtt. Így a duplikált egyes sávot sem tünteti el.

4.4 Félsávok

Mit értünk tulajdonképpen a félsáv fogalom alatt, és milyen jelentősége lehet a védelem szempontjából?

A fogalom megértéséhez tudnunk kell, hogy a lemezegység író/olvasó feje az ismert egy sáv méretű lépéstávolságon kívül a félsávnyi távolságot képes átlépni. Sőt a félsávok tartalmát probléma nélkül vissza is tudja olvasni, szemben felírásukkal, ami már nem nevezhető kifejezetten egyszerűnek.

A magyarázat kézenfekvő: a 1541-es lemezegységet tulajdonképpen 80 sáv kezelésére készítették fel. Pillanatnyilag ebből a Commodore 64-es csak 40 sávot használ fel, ezért az egységbe 40 sávot kezelni képes írófejet építettek be, az eredetileg 80 sávra tervezett olvasófejet viszont megtartották.

A félsávokra pozicionálva, szélesebb írófej írás közben felülírja a szomszédos sáv egy részét is. Ez a magyarázata annak, hogy a 1541-es lemezegységgel három egymást követő félsávot nem lehet hibátlanul teleírni.

Ezt a tényt már számos szoftverház felhasználta programtermékei védelmére. A védelem lényege az, hogy egy másik lemezegység igénybevételével felírnak a lemezre három egymást követő félsávot, és ezeket a program elindításakor a VC1541-es egység ellenőrzi.

A felírás a következőképpen oldható meg: először három egymást követő félsávról töröljük az összes szinkronjelet, majd az elsőre egyetlen szinkronjelet és mögé néhány adatot írunk. Ezután a fejet hirtelen a szomszédos félsávra mozgatjuk, és arra is felírjuk az adatokat a szinkronjel után. Végül a műveletet a harmadik félsávra is regisztrálhatjuk. Az egymást követő három felíró művelet közben a fejet olyan hirtelen mozdítjuk el, hogy a lemez nem tud tökéletesen megfordulni. Írás közben ugyan a szomszédos félsávokat is felülírjuk, de csak azokon a helyeken, ahol a kijelölt félsávokon nincs adat. Mindez csak akkor lehetséges, ha a felíráshoz szükséges időtartam alatt a lemez összesen egyszer fordul körbe.

A másolóprogramok egyszerre mindig egy teljes sávot írnak és olvasnak. Eközben a szomszédos félsávok adattartalmát természetesen megsértik. Tehát a másolást még akkor sem „tudják" eredményesen végrehajtani, ha „tudják", hogy milyen védelemmel állnak szemben; azt semmiképpen nem ismerik fel, hogy egy adott fordulaton belül hol kell félsávra váltani.

A védelmet létrehozó program listája:

```
10 OPEN1,8,15,"I"
20 READ HD$:IF VAL(HD$)=-1THEN100
23 GOSUB 5260
24 PRINTN,HD$
25 SU=SU+D
30 PRINT#1,"M-W"CHR$(N)CHR$(5)CHR$(1)CHR$(D)
```

```
40 N=N+1:GOTO 20
100 IFSU<>20644THEN PRINT"H1BAS DATA":STOP
130 PRINT#1,"M-E"CHR$(3)CHR$(5)
135 FORN=1TO 500:NEXT
140 FORN=1TO 500:NEXT
145 PRINT#1,"M-R"CHR$(2)CHR$(0)CHR$(1)
150 GET#1,A$:A=ASC(A$+CHR$(0))
160 IFA>127 THEN 145
170 IFA=1 THEN PRINT"OK":END
180 PRINT"H1BA":END
500 DATA 4C,19,05:REM        JMP   0519
510 DATA 20,42,D0:REM        JSR   D042
520 DATA A9,27   :REM        LDA  #27
530 DATA 85,0A   :REM        STA   0A
540 DATA A9,00   :REM        LDA  #00
550 DATA 85,0B   :REM        STA   0B
560 DATA A9,E0   :REM        LDA  #E0
570 DATA 85,02   :REM        STA   02
580 DATA 24,02   :REM        BIT   02
590 DATA 30,FC   :REM        BMI   0512
600 DATA 4C,42,D0:REM        JMP   D042
610 DATA AD,00,1C:REM        LDA   1C00
620 DATA 29,9F   :REM        AND  #9F
630 DATA 09,08   :REM        ORA  #08
640 DATA 8D,00,1C:REM        STA   1C00
650 DATA 20,0E,FE:REM        JSR   FE0E
660 DATA 20,00,FE:REM        JSR   FE00
670 DATA A9,02   :REM        LDA  #02
680 DATA 85,3B   :REM        STA   3B
690 DATA 20,A0,05:REM        JSR   05A0
700 DATA 20,0E,FE:REM        JSR   FE0E
710 DATA 20,00,FE:REM        JSR   FE00
720 DATA C6,3B   :REM        DEC   3B
730 DATA D0,F3   :REM        BNE   052D
740 DATA AD,0C,1C:REM        LDA   1C0C
750 DATA 29,1F   :REM        AND  #1F
760 DATA 09,C0   :REM        ORA  #C0
770 DATA 8D,0C,1C:REM        STA   1C0C
780 DATA A9,FF   :REM        LDA  #FF
790 DATA 8D,03,1C:REM        STA   1C03
800 DATA 8D,01,1C:REM        STA   1C01
810 DATA A2,C8   :REM        LDX  #C8
820 DATA 50,FE   :REM        BVC   054E
830 DATA B8      :REM        CLV
840 DATA CA      :REM        DEX
850 DATA D0,FA   :REM        BNE   054E
860 DATA BD,80,05:REM        LDA   0580,X
870 DATA 50,FE   :REM        BVC   0557
880 DATA B8      :REM        CLV
890 DATA 8D,01,1C:REM        STA   1C01
900 DATA E8      :REM        INX
910 DATA E0,08   :REM        CPX  #08
920 DATA D0,F2   :REM        BNE   0554
930 DATA 50,FE   :REM        BVC   0562
940 DATA B8      :REM        CLV
950 DATA 20,00,FE:REM        JSR   FE00
960 DATA 20,A3,05:REM        JSR   05A3
970 DATA AD,55,05:REM        LDA   0555
980 DATA 18      :REM        CLC
990 DATA 69,08   :REM        ADC  #08
```

```
1000 DATA 8D,55,05:REM        STA   0555
1010 DATA C9,98   :REM        CMP  #98
1020 DATA D0,C2   :REM        BNE   053A
1030 DATA 20,A0,05:REM        JSR   05A0
1040 DATA 4C,9E,FD:REM        JMP   FD9E
5000 DATA 28,00
5010 DATA 69,A5,5A,96,56,59,A6,A9
5020 DATA 59,9A,6A,65,66,55,99,AA
5030 DATA 66,95,96,69,A5,5A,6A,6A
5040 DATA 00,00,00,00,00,00,00,00
5050 DATA A9,CA   :REM        LDA  #CA
5060 DATA 2C      :REM        BYTE  2C
5070 DATA A9,E8   :REM        LDA  #E8
5080 DATA 8D,AB,05:REM        STA   05AB
5090 DATA AE,00,1C:REM        LDX   1C00
5100 DATA CA      :REM        DEX
5110 DATA 8A      :REM        TXA
5120 DATA 29,03   :REM        AND  #03
5130 DATA 85,4B   :REM        STA   4B
5140 DATA AD,00,1C:REM        LDA   1C00
5150 DATA 29,FC   :REM        AND  #FC
5160 DATA 05,4B   :REM        ORA   4B
5170 DATA 8D,00,1C:REM        STA   1C00
5180 DATA A2,10   :REM        LDX  #10
5190 DATA A0,00   :REM        LDY  #00
5200 DATA 88      :REM        DEY
5210 DATA D0,FD   :REM        BNE   05E2
5220 DATA CA      :REM        DEX
5230 DATA D0,FA   :REM        BNE   05E2
5240 DATA 60      :REM        RTS
5250 DATA -1      :REM  PROGRAM VEGE
5260 REM HEXABOL DEC.-BE
5270 H$="0123456789ABCDEF"
5280 IF LEN(HD$)=0 THEN PRINT"HIBA":STOP
5290 IF LEN(HD$)<> 2 THEN HD$=LEFT$(HD$,2)
5300 H=0
5310 FOR ZZ=1 TO 16
5320 IF LEFT$(HD$,1)=MID$(H$,ZZ,1) THEN H=ZZ:GOTO 5340
5330 NEXT ZZ
5340 H=H-1
5350 L=0
5360 FOR ZZ=1 TO 16
5370 IF RIGHT$(HD$,1)=MID$(H$,ZZ,1) THEN L=ZZ:GOTO 5390
5380 NEXT ZZ
5390 L=L-1
5400 D=H*16+L
5410 RETURN
READY.
```

A programot a $503-as címről indítjuk, s a védelmet a 35-ös sáv fölé írjuk fel. A lekérdező programot szintén erről a címről indítottuk el, s a lekérdezés eredményét most is a RAM $10-es címére helyeztük. Ha a cím tartalma 00, másolási kísérletet észleltünk.

```
6 REM      FELSAVOK
10 OPEN1,8,15,"I"
20 READ HD$:IF VAL(HD$)=-1THEN100
21 PRINTN,HD$
22 GOSUB 5000
```

```
25 SU=SU+D
30 PRINT#1,"M-W"CHR$(N)CHR$(5)CHR$(1)CHR$(D)
40 N=N+1:GOTO 20
100 IFSU<>24843THEN PRINT"HIBAS DATA":STOP
135 PRINT#1,"M-E"CHR$(3)CHR$(5)
140 FORN=1TO 500:NEXT
145 PRINT#1,"M-R"CHR$(16)CHR$(0)CHR$(1)
150 GET#1,A$:A=ASC(A$+CHR$(0))
170 IFA=255THEN PRINT"OK.....":END
180 PRINT"HIBAS":END
400 DATA 4C,19,05:REM        JMP   0519
410 DATA 20,42,D0:REM        JSR   D042 LEMEZ INIT.
420 DATA A9,27    :REM        LDA  #27   SAVSZAM
430 DATA 85,0A    :REM        STA   0A
440 DATA A9,00    :REM        LDA  #00   SZEKTORSZAM
450 DATA 85,0B    :REM        STA   0B
460 DATA A9,E0    :REM        LDA  #E0   JOB KOD
470 DATA 85,02    :REM        STA   02
480 DATA A9,02    :REM  C1    LDA   02
490 DATA 30,FC    :REM        BMI   C1   VARAKOZAS
500 DATA 4C,42,D0:REM        JMP   D042 LEMEZ INIT.
510 DATA AD,00,1C:REM        LDA   1C00 ADATKAPU
520 DATA 29,9F    :REM        AND  #9F
530 DATA 09,08    :REM        ORA  #08
540 DATA 8D,00,1C:REM        STA   1C00
550 DATA EA       :REM        NOP
560 DATA EA       :REM        NOP
570 DATA 20,93,05:REM        JSR   0593
580 DATA EA       :REM        NOP
590 DATA A9,00    :REM        LDA  #00
600 DATA 85,3B    :REM        STA   3B
610 DATA AD,00,1C:REM        LDA   1C00
620 DATA 10,FB    :REM        BPL   052D
630 DATA AD,00,1C:REM        LDA   1C00
640 DATA 10,0D    :REM        BPL   0544
650 DATA EE,FE,05:REM        INC   05FE
660 DATA D0,F6    :REM        BNE   0532
670 DATA EE,FF,05:REM        INC   05FF
680 DATA D0,F1    :REM        BNE   0532
690 DATA 4C,8F,05:REM        JMP   058F
700 DATA AD,01,1C:REM        LDA   1C01
710 DATA B8       :REM        CLV
720 DATA A2,00    :REM        LDX  #00
730 DATA 50,FE    :REM        BVC   054A
750 DATA B8       :REM        CLV
760 DATA AD,01,1C:REM        LDA   1C01
770 DATA DD,A0,05:REM        CMP   05A0,X
780 DATA D0,32    :REM        BNE   0587
790 DATA E8       :REM        INX
800 DATA E0,06    :REM        CPX  #06
810 DATA D0,F0    :REM        BNE   054A
820 DATA 20,C3,05:REM        JSR   05C3
830 DATA A9,3F    :REM        LDA  #3F
840 DATA 85,3B    :REM        STA   3B
850 DATA 20,93,05:REM        JSR   0593
860 DATA EA       :REM        NOP
870 DATA EA       :REM        NOP
890 DATA EA       :REM        NOP
900 DATA AD,51,05:REM        LDA   0551
910 DATA 18       :REM        CLC
```

```
920 DATA 69,08    :REM        ADC  #08
930 DATA 8D,51,05:REM         STA   0551
940 DATA C9,B8    :REM        CMP  #B8
950 DATA D0,BE    :REM        BNE   0532
960 DATA 20,C0,05:REM         JSR   05C0
970 DATA 20,C0,05:REM         JSR   05C0
980 DATA 20,C0,05:REM         JSR   05C0
990 DATA A9,FF    :REM        LDA  #FF
1000 DATA 8D,10,00:REM         STA   0010
1010 DATA 8D,11,00:REM         STA   0011
1020 DATA D0,BA    :REM        BNE   0541
1030 DATA E6,3B    :REM        INC   3B
1040 DATA A5,3B    :REM        LDA   3B
1050 DATA C9,40    :REM        CMP  #40
1060 DATA D0,A3    :REM        BNE   0532
1070 DATA A9,00    :REM        LDA  #00
1080 DATA F0,EC    :REM        BEQ   057F
1090 DATA A9,00    :REM        LDA  #00
1100 DATA 8D,FE,05:REM         STA   05FE
1120 DATA 8D,FF,05:REM         STA   05FF
1130 DATA 60       :REM        RTS
1140 DATA 00,00,00,00
1150 DATA 69,A5,5A,96,56,59,A6,A9
1160 DATA 59,9A,6A,65,66,55,99,AA
1170 DATA 66,95,96,69,A5,5A,6A,6A
1180 DATA 00,00,00,00,00,00,00,00
2000 DATA A9,CA    :REM        LDA  #CA
2010 DATA 2C       :REM        BYTE  2C
2012 DATA A9,E8    :REM        LDA  #E8
2020 DATA 8D,AB,05:REM         STA   05AB
2030 DATA AE,00,1C:REM         LDX   1C00
2040 DATA CA       :REM        DEX
2050 DATA 8A       :REM        TXA
2060 DATA 29,03    :REM        AND  #03
2070 DATA 85,4B    :REM        STA   4B
2080 DATA AD,00,1C:REM         LDA   1C00
2090 DATA 29,FC    :REM        AND  #FC
2100 DATA 05,4B    :REM        ORA   4B
2110 DATA 8D,00,1C:REM         STA   1C00
2120 DATA A2,10    :REM        LDX  #10
2130 DATA A0,00    :REM        LDY  #00
2140 DATA 88       :REM        DEY
2150 DATA D0,FD    :REM        BNE   05E2
2160 DATA CA       :REM        DEX
2170 DATA D0,FA    :REM        BNE   05E2
2180 DATA 60       :REM        RTS
2190 DATA -1       :REM  PROGRAM VEGE
5000 REM HEXABOL DEC.-BE
5010 H$="0123456789ABCDEF"
5020 IF LEN(HD$)=0 THEN PRINT"H1BA":STOP
5030 IF LEN(HD$)<> 2 THEN HD$=LEFT$(HD$,2)
5040 H=0
5050 FOR ZZ=1 TO 16
5060 IF LEFT$(HD$,1)=MID$(H$,ZZ,1) THEN H=ZZ:GOTO 5080
5070 NEXT ZZ
5080 H=H-1
5090 L=0
5100 FOR ZZ=1 TO 16
5110 IF RIGHT$(HD$,1)=MID$(H$,ZZ,1) THEN L=ZZ:GOTO 5130
5120 NEXT ZZ
```

```
5130 L=L-1
5140 D=H*16+L
5150 RETURN
READY.
```

4.5 Nullával teleírt sávok

Ezzel a megoldással létrehozott védelmet rendkívül nehéz feltörni, a másolás ui. az adatok visszaolvasása és felírása a lemezegység hardverfeltételei mellett szinte lehetetlen. Az is komoly nehézségekbe ütközik, hogy a másolóprogram a védelmet azonosítsa!

Tekintsük át röviden az adatok felírásának technikáját. A $1C01 címen tárolt, felírásra előkészített adatokat az író/olvasó fej bitenként átveszi és mágnesjelekké alakítja.

Az 1-es bitnek valójában váltakozó, a 0-s bitnek pedig állandó mágneses mező felel meg. Hogyan lehet a 0-s bitet egyáltalán felismerni, ha felírása közben „semmi sem történik"? A biteket a rendszer egy timer segítségével azonosítja, amelynek lefutási ideje pontosan megfelel egy bit felírási, ill. visszaolvasási idejének. Ha a timer lefutása közben a rendszer felismeri a mágneses mező irányváltozását, a beolvasott bitet 1-nek tekinti, és a timert újraindítja. Ha a timer lefutása közben nem volt irányváltozás, akkor a bit értéke 0.

A lemezegység szinkronhibái ilyen technikai megoldás mellett könnyen olvasási hibához vezethetnének, ha több egymást követő 0 értékű bitet kellene azonosítani. A vezérlőnek ui. tökéletesen a timerre kellene hagyatkoznia, a lemez semmilyen támpontot nem nyújthatna. Ez a magyarázata annak, hogy szükség volt a GCR kódrendszerre, amelyben kettőnél több nulla bit nem állhat egymás mellett. Azt a tényt, hogy a vezérlő túl sok nulla bitet találva „zavarba jön", felhasználhatjuk egy másolásvédelmi eljárás kialakítására. Az egyik blokk közepére felírunk egymás mögé néhány nulla bitet, ami azt jelenti, hogy ezen a szakaszon a mágnesréteg változatlan lesz. A vezérlő ezt az adatterületet nem tudja feldolgozni és valahogyan „kibetűzni". Eközben a bitek közé beilleszt néhány nem létező 1-est. Amikor az olvasást megismétli, a már beolvasott adatok tökéletesen el fognak térni az újonnan olvasottaktól. A másolóprogram ezek után nem tehet mást, mint azt, hogy a valódi adatok helyett azokat tárolja, amelyeket a vezérlő „fantáziája" hozott létre. Ha a másolatot többször visszaolvassuk, az eredmény mindig azonos lesz, elvileg hibátlan, hiszen a vezérlő olyan adatokat írt a lemezre, amelyeket tud értelmezni. A módszer tehát elég ígéretesnek látszik ahhoz, hogy érdemes legyen foglalkozni vele.

Térjünk most rá a gyakorlati megvalósításra. A következő program létrehozza a hibás adatblokkot a kijelölt szektorban. A programot a $0552-es címről indítjuk:

```
5 REM NULLAS SAV IRO
10 OPEN1,8,15,"I"
20 READ HD$:IF VAL(HD$)=-1 THEN100
22 GOSUB 1000
23 PRINTN,HD$
25 SU=SU+D
30 PRINT#1,"M-W"CHR$(N)CHR$(5)CHR$(1)CHR$(D)
40 N=N+1:GOTO20
100 IFSU<>12308 THEN PRINT"HIBAS DATA":STOP
105 INPUT"SAV";T
110 INPUT"SZEKTOR";S
120 PRINT#1,"M-W"CHR$(83)CHR$(5)CHR$(1)CHR$(T)
125 PRINT#1,"M-W"CHR$(87)CHR$(5)CHR$(1)CHR$(S)
135 PRINT#1,"M-E"CHR$(82)CHR$(5)
140 FORN=1TO500:NEXT
145 PRINT#1,"M-R"CHR$(2)CHR$(0)CHR$(1)
150 GET#1,A$:A=ASC(A$+CHR$(0))
160 IF A>127 THEN 145
170 IF A=1THEN PRINT"OK":END
180 PRINT"HIBA":END
100 DATA 20,10,F5:REM        JSR   F510
110 DATA A2,09    :REM        LDX   #09
120 DATA 50,FE    :REM        BVC    0505
130 DATA B8       :REM        CLV
140 DATA CA       :REM        DEX
150 DATA D0,FA    :REM        BNE    0505
160 DATA A9,FF    :REM        LDA   #FF
170 DATA 8D,03,1C:REM         STA    1C03
180 DATA AD,0C,1C:REM         LDA    1C0C
190 DATA 29,1F    :REM        AND   #1F
200 DATA 09,C0    :REM        ORA   #C0
210 DATA 8D,0C,1C:REM         STA    1C0C
220 DATA A9,FF    :REM        LDA   #FF
230 DATA A2,05    :REM        LDX   #05
240 DATA 8D,01,1C:REM         STA    1C01
250 DATA B8       :REM        CLV
260 DATA 50,FE    :REM        BVC    0522
270 DATA B8       :REM        CLV
280 DATA CA       :REM        DEX
290 DATA D0,FA    :REM        BNE    0522
300 DATA A9,55    :REM        LDA   #55
310 DATA 8D,01,1C:REM         STA    1C01
320 DATA 50,FE    :REM        BVC    052D
330 DATA B8       :REM        CLV
340 DATA A9,CE    :REM        LDA   #CE
350 DATA 8D,01,1C:REM         STA    1C01
360 DATA 50,FE    :REM        BVC    0535
370 DATA B8       :REM        CLV
380 DATA E8       :REM        INX
390 DATA E0,20.   :REM        CPX   #20
400 DATA D0,F8    :REM        BNE    0535
410 DATA A9,00    :REM        LDA   #00
420 DATA A2,20    :REM        LDX   #20
430 DATA 8D,01,1C:REM         STA    1C01
440 DATA 50,FE    :REM        BVC    0544
450 DATA B8       :REM        CLV
460 DATA CA       :REM        DEX
470 DATA D0,FA    :REM        BNE    0544
780 DATA 20,00,FE:REM         JSR    FE00
790 DATA A9,01    :REM        LDA   #01
```

```
800 DATA 4C,69,F9:REM        JMP   F969
810 DATA A9,12   :REM        LDA  #12
820 DATA 85,0A   :REM        STA   0A
830 DATA A9,00   :REM        LDA  #00
840 DATA 85,0B   :REM        STA   0B
850 DATA A9,E0   :REM        LDA  #E0
860 DATA 85,02   :REM        STA   02
870 DATA A5,02   :REM        LDA   02
880 DATA 30,FC   :REM        BMI   055E
890 DATA 60      :REM        RTS
900 DATA -1      ;REM PROGRAM VEGE
1000 REM HEXABOL DEC.-BE
1010 H$="0123456789ABCDEF"
1020 IF LEN(HD$)=0 THEN PRINT"HIBA":STOP
1030 IF LEN(HD$)<> 2 THEN HD$=LEFT$(HD$,2)
1040 H=0
1050 FOR ZZ=1 TO 16
1060 IF LEFT$(HD$,1)=MID$(H$,ZZ,1) THEN H=ZZ:GOTO 1080
1070 NEXT ZZ
1080 H=H-1
1090 L=0
1100 FOR ZZ=1 TO 16
1110 IF RIGHT$(HD$,1)=MID$(H$,ZZ,1) THEN L=ZZ:GOTO 1130
1120 NEXT ZZ
1130 L=L-1
1140 D=H*16+L
1150 RETURN
READY.
```

A nulla bitek elé olyan adatokat írtunk, hogy 22-es Read Error ne keletkezhessen. A bitképben az első helyen, a szinkronjel után, $07 (az adatblokk azonosítója) áll.

A védelem lekérdezése létrehozásánál is egyszerűbb. A védett blokkot egymás után kétszer olvassuk be, és a beolvasott adatokat összehasonlítjuk. Ha nem egyeznek, a lemez eredeti. A vizsgálat eredményének kódját most is a $10-es címen keresztül adjuk át a gépnek. Ha a kód $FF, a lemez másolat, ha pedig $100, akkor a lemez eredeti.

A lemezprogramot a $0500-as címtől kezdve helyeztük el, és a $0543-ról indítjuk:

```
10 OPEN1,8,15,"1"
20 READ HD$:IF VAL(HD$)=-1 THEN100
24 GOSUB 1000
25 SU=SU+D
26 PRINTN,HD$
30 PRINT#1,"M-W"CHR$(N)CHR$(5)CHR$(1)CHR$(D)
40 N=N+1:GOTO20
100 IFSU<>9963  THEN PRINT"HIBAS DATA":STOP
105 INPUT"SAV";T
110 INPUT"SZEKTOR";S
120 PRINT#1,"M-W"CHR$(68)CHR$(5)CHR$(1)CHR$(T)
125 PRINT#1,"M-W"CHR$(72)CHR$(5)CHR$(1)CHR$(S)
135 PRINT#1,"M-E"CHR$(67)CHR$(5)
140 FORN=1TO500:NEXT
145 PRINT#1,"M-R"CHR$(16)CHR$(0)CHR$(1)
150 GET#1,A$:A=ASC(A$+CHR$(0))
```

```
170 IF A=255THEN PRINT"OK ":END
180 PRINT"HIBAS ":END
300 DATA A9,05    :REM        LDA  #05
310 DATA 85,37    :REM        STA   37
320 DATA 20,0A,F5:REM        JSR  F50A
330 DATA A2,00    :REM        LDX  #00
340 DATA 50,FE    :REM        BVC   0509
350 DATA B8       :REM        CLV
360 DATA AD.01,1C:REM        LDA   1C01
370 DATA 9D,00,03:REM        STA   0300,X
380 DATA E8       :REM        INX
390 DATA D0,F4    :REM        BNE   0508
400 DATA 20.0A.F5:REM        JSR  F50A
410 DATA A2,00    :REM        LDX  #0Ø
420 DATA 50,FE    :REM        BVC   051A
430 DATA B8       :REM        CLV
440 DATA AD,01.1C:REM        LDA   1C01
450 DATA 9D,00,04:REM        STA   0400,X
460 DATA E8       :REM        INX
470 DATA D0,F4    :REM        BNE   051A
480 DATA BD,00.03:REM        LDA   0300,X
490 DATA DD,00,04:REM        CMP   0400,X
500 DATA D0,0E    :REM        BNE   053C
510 DATA E8       :REM        INX
520 DATA D0,F5    :REM        BNE   0526
530 DATA C6,37    :REM        DEC   37
540 DATA D0,CF    :REM        BNE   0504
550 DATA A9,00    :REM        LDA  #00
560 DATA 85.10    :REM        STA   10
570 DATA 4C,9E,FD:REM        JMP   FD9E
580 DATA A9,FF    :REM        LDA  #FF
590 DATA 85,10    :REM        STA   10
600 DATA 4C,9E.FD:REM        JMP   FD9E
610 DATA A9,01    :REM        LDA  #01
620 DATA 85.0A    :REM        STA   0A
630 DATA A9,00    :REM        LDA  #00
640 DATA 85,0B    :REM        STA   0B
650 DATA A9,E0    :REM        LDA  #E0
660 DATA 85.02    :REM        STA   02
670 DATA A5,02    :REM        LDA   02
680 DATA 30,FC    :REM        BMI   054F
690 DATA 60       :REM        RTS
700 DATA -1       :REM  PROGRAM VEGE
1000 REM HEXABOL DEC.-BE
1010 H$="0123456789ABCDEF"
1020 IF LEN(HD$)=0 THEN PRINT"HIBA":STOP
1030 IF LEN(HD$)<> 2 THEN HD$=LEFT$(HD$,2)
1040 H=0
1050 FOR ZZ=1 TO 16
1060 IF LEFT$(HD$,1)=MID$(H$,ZZ,1) THEN H=ZZ:GOTO 1080
1070 NEXT ZZ
1080 H=H-1
1090 L=0
1100 FOR ZZ=1 TO 16
1110 IF RIGHT$(HD$,1)=MID$(H$,ZZ,1) THEN L=ZZ:GOTO 1130
1120 NEXT ZZ
1130 L=L-1
1140 D=H*16+L
1150 RETURN
READY.
```

4.6 A teljes sáv átvizsgálása

Az, hogy egy sáv összes byte-ját egyenként átvizsgáljuk, kiindulópontja lehet egy védelmi eljárásnak. Annak, hogy miért nem lehet egy sávot egy programmal lemásolni, egyszerű a magyarázata. Mivel a sávok nem egyenlő hosszúak, a felírható byte-ok száma változó. Az egy sávon belüli eltérés az egység motorjának egyenetlenségeiből adódik.

A másolás további akadálya az, hogy az utolsó adatbyte közvetlenül a szinkronjel mellett áll. A védelem akkor is eredményes lesz, ha a másolóprogramnak esetleg már minden adatot sikerült beolvasnia. Ha ui. az adatok végét jelző szinkronbyte-ba további nulla biteket írunk, ami az író/olvasó fej olvasásra kapcsolása közben könnyen megoldható, a másoló az előző alfejezetben leírtak szerint jár el. A lekérdezés tehát itt is egyértelműen azonosítja a másolatot.

A lekérdező programot arra az eshetőségre is fel kell készítenünk, hogy a szinkronjel megsérül, amikor a fejet olvasó üzemmódra kapcsoljuk. A megoldás az, hogy több írási kísérletet hajtunk végre. Az írási kísérleteket mindaddig ismételjük, amíg a másolásvédelem rögzítését már semmi nem gátolja.

Most nézzük meg részletesen, ill. hogy hogyan dolgozik az olvasó-, ill. felíró program. A védelem létrehozásának első lépéseként a megadott sávot töröljük. Ezután az adatok felírása következik, ami kb. hatezer $45 és $79 értékű byte rögzítésével egyenértékű. Az adatok végét $35, $66 jelzi, amely után ismét egy szinkronjel áll.

A lekérdező program vár a szinkronjelre, majd beolvassa az adatokat. A beolvasott adatok számát folyamatosan tárolja, és figyeli, hogy elérte-e már a $35 kódot. Ha igen, beolvas további hat byte-ot. Ezek közül az első $66. Ha a szinkronjel hibátlan, kezdetét $FF jelzi, ami alapján a rendszer hardveresen megakadályozza a további olvasást, ill. átlép a következő adatblokk első byte-jára. Ha a szinkronjelet az író/olvasó fej olvasásra kapcsolásával tönkretettük, a következő adatblokk azonosítása előtt álló, oda nem illő byte a másolását megakadályozza.

Az elmondottak szerint készült a következő program, amely a $0571-es címen (START) indul, és létrehozza a lemezen a védelmet. A KERD címkétől ($0543) lévő programrésszel ellenőrizhetjük, hogy a szinkronjel felírása az író/olvasó fej átkapcsolása közben eredményes volt-e.

```
10 OPEN1,8,15,"I"
20 READ HD$:IF VAL(HD$)=-1 THEN100
22 GOSUB 1000
25 SU=SU+D
27 PRINT N,HD$
30 PRINT#1,"M-W"CHR$(N)CHR$(5)CHR$(1)CHR$(D)
40 N=N+1:GOTO20
```

```
100 IFSU<>15481 THEN PRINT"H1BAS DATA":STOP
110 INPUT"SAV";T
120 PRINT#1,"M-W"CHR$(114)CHR$(5)CHR$(1)CHR$(T
125 PRINT#1,"M-W"CHR$(72)CHR$(5)CHR$(1)CHR$(S)
130 PRINT#1,"M-E"CHR$(113)CHR$(5)
140 PRINT#1,"I"
150 CLOSE1
400 DATA AD,00,1C:REM        LDA   1C00
410 DATA 29,9F   :REM        AND  #9F
420 DATA 09,08   :REM        ORA  #08
430 DATA 8D,00,1C:REM        STA   1C00
440 DATA 20,A3,FD:REM        JSR   FDA3
450 DATA A2,02   :REM        LDX  #02
460 DATA A0,18   :REM        LDY  #18
470 DATA A9,45   :REM        LDA  #45
480 DATA 8D,01,1C:REM        STA   1C01
490 DATA 50,FE   :REM        BVC   0516
500 DATA B8      :REM        CLV
510 DATA E8      :REM        INX
520 DATA A9,79   :REM        LDA  #79
530 DATA 8D,01,1C:REM        STA   1C01
540 DATA 50,FE   :REM        BVC   051F
550 DATA B8      :REM        CLV
560 DATA E8      :REM        INX
570 DATA D0,EC   :REM        BNE   0511
580 DATA 88      :REM        DEY
590 DATA D0,E9   :REM        BNE   0511
600 DATA A9,35   :REM        LDA  #35
610 DATA 8D,01,1C:REM        STA   1C01
620 DATA 50,FE   :REM        BVC   052D
630 DATA B8      :REM        CLV
640 DATA A9,66   :REM        LDA  #66
650 DATA 8D,01,1C:REM        STA   1C01
660 DATA 50,FE   :REM        BVC   0535
670 DATA B8      :REM        CLV
680 DATA A9,FF   :REM        LDA  #FF
690 DATA 8D,01,1C:REM        STA   1C01
700 DATA 50,FE   :REM        BVC   053D
710 DATA B8      :REM        CLV
720 DATA 20,00,FE:REM        JSR   FE00
730 DATA AD,00,1C:REM KERD   LDA   1C00
740 DATA 30,FB   :REM        BMI   0543
750 DATA AD,01,1C:REM        LDA   1C01
760 DATA B8      :REM        CLV
770 DATA 50,FE   :REM        BVC   054C
780 DATA B8      :REM        CLV
790 DATA AD,01,1C:REM        LDA   1C01
800 DATA C9,35   :REM        CMP  #35
810 DATA D0,F6   :REM        BNE   054C
820 DATA A2,00   :REM        LDX  #00
830 DATA 50,FE   :REM        BVC   0558
840 DATA B8      :REM        CLV
850 DATA AD,01,1C:REM        LDA   1C01
860 DATA DD,6B.05:REM        CMP   056B,X
870 DATA D0,A7   :REM        BNE   050A
880 DATA E8      :REM        INX
890 DATA E0,06   :REM        CPX  #06
900 DATA D0,F0   :REM        BNE   0558
910 DATA 4C,9E,FD:REM        JMP   FD9E
920 DATA 66,FF,45,79,45,79
```

```
940 DATA A9,01    :REM START LDA  #01
950 DATA 85,0A    :REM       STA   0A
960 DATA A9,E0    :REM       LDA  #E0
970 DATA 85,02    :REM       STA   02
980 DATA A5,02    :REM       LDA   02
990 DATA 30,FC    :REM       BMI   0579
994 DATA 60       :REM       RTS
996 DATA -1       REM PROGRAM VEGE
1000 REM HEXABOL DEC.-BE
1010 H$="0123456789ABCDEF"
1020 IF LEN(HD$)=0 THEN PRINT"HIBA":STOP
1030 IF LEN(HD$)<> 2 THEN HD$=LEFT$(HD$,2)
1040 H=0
1050 FOR ZZ=1 TO 16
1060 IF LEFT$(HD$,1)=MID$(H$,ZZ,1) THEN H=ZZ:GOTO 1080
1070 NEXT ZZ
1080 H=H-1
1090 L=0
1100 FOR ZZ=1 TO 16
1110 IF RIGHT$(HD$,1)=MID$(H$,ZZ,1) THEN L=ZZ:GOTO 1130
1120 NEXT ZZ
1130 L=L-1
1140 D=H*16+L
1150 RETURN
READY.
```

A védelmet ellenőrző program listája:

```
10 OPEN1,8,15,"I"
20 READ HD$:IF VAL(HD$)=-1 THEN100
22 GOSUB 1000
25 SU=SU+D
27 PRINTN,HD$
30 PRINT#1,"M-W"CHR$(N)CHR$(6)CHR$(1)CHR$(D)
40 N=N+1:GOTO20
100 IFSU<>13657 THEN PRINT"HIBA AZ ADATOKBAN":STOP
110 INPUT"SAV";T
120 PRINT#1,"M-W"CHR$(98)CHR$(6)CHR$(1)CHR$(T)
130 PRINT#1,"M-E"CHR$(97)CHR$(6)
140 FORN=1TO500:NEXT
150 PRINT#1,"M-R"CHR$(16)CHR$(0)CHR$(1)
160 GET#1,A$
170 IFASC(A$+CHR$(0))<>255THEN PRINT"HIBA";
175 PRINT ASC(A$+CHR$(0)):END
180 PRINT"A VEDELEM RENDBEN"
200 END
400 DATA AD,00,1C:REM       LDA   1C00
410 DATA 29,9F    :REM       AND  #9F
420 DATA 09,08    :REM       ORA  #08
430 DATA 8D,00,1C:REM       STA   1C00
440 DATA A2,00    :REM       LDX  #00
450 DATA A0,00    :REM       LDY  #00
460 DATA AD,00,1C:REM       LDA   1C00
470 DATA 30,FB    :REM       BMI   060E
480 DATA AD,01,1C:REM       LDA   1C01
490 DATA B8       :REM       CLV
500 DATA 50,FE    :REM       BVC   0617
510 DATA B8       :REM       CLV
520 DATA AD,01,1C:REM       LDA   1C01
530 DATA C9,45    :REM       CMP  #45
```

```
540 DATA D0,11    :REM          BNE   0632
550 DATA E8       :REM          INX
560 DATA 50,FE    :REM          BVC   0622
570 DATA B8       :REM          CLV
580 DATA AD,01,1C:REM           LDA   1C01
590 DATA C9,79    :REM          CMP   #79
600 DATA D0,06    :REM          BNE   0632
610 DATA E8       :REM          INX
620 DATA D0,E8    :REM          BNE   0617
630 DATA C8       :REM          INY
640 DATA D0,E5    :REM          BNE   0617
650 DATA C9,35    :REM          CMP   #35
660 DATA D0,21    :REM          BNE   0657
670 DATA E0,FE    :REM          CPX   #FE
680 DATA D0,1D    :REM          BNE   0657
690 DATA C0,17    :REM          CPY   #17
700 DATA D0,19    :REM          BNE   0657
710 DATA A2,00    :REM          LDX   #00
720 DATA 50,FE    :REM          BVC   0640
730 DATA B8       :REM          CLV
740 DATA AD,01,1C:REM           LDA   1C01
750 DATA DD,5B,06:REM           CMP   065B,X
760 DATA D0,0C    :REM          BNE   0657
770 DATA E8       :REM          INX
780 DATA E0,06    :REM          CPX   #06
790 DATA D0,F0    :REM          BNE   0640
800 DATA A9,FF    :REM          LDA   #FF
810 DATA 85,10    :REM          STA   10
820 DATA 4C,9E,FD:REM           JMP   FD9E
830 DATA A9,00    :REM          LDA   #00
840 DATA F0,F7    :REM          BEQ   0652
850 DATA 66,FF,45,79,45,79
860 DATA A9,02    :REM        LDA   #02
870 DATA 85,0C    :REM        STA   0C
880 DATA A9,E0    :REM        LDA   #E0
890 DATA 85,03    :REM        STA   03
900 DATA A5,03    :REM        LDA   03
910 DATA 30,FC    :REM        BMI   0669
920 DATA 60       :REM        RTS
930 DATA -1       :REM  PROGRAM VEGE
1000 REM HEXABOL DEC.-BE
1010 H$="0123456789ABCDEF"
1020 IF LEN(HD$)=0 THEN PRINT"HIBA":STOP
1030 IF LEN(HD$)<> 2 THEN HD$=LEFT$(HD$,2)
1040 H=0
1050 FOR ZZ=1 TO 16
1060 IF LEFT$(HD$,1)=MID$(H$,ZZ,1) THEN H=ZZ:GOTO 1080
1070 NEXT ZZ
1080 H=H-1
1090 L=0
1100 FOR ZZ=1 TO 16
1110 IF RIGHT$(HD$,1)=MID$(H$,ZZ,1) THEN L=ZZ:GOTO 1130
1120 NEXT ZZ
1130 L=L-1
1140 D=H*16+L
1150 RETURN
READY.
```

Előfordulhat, hogy a védelem létrehozása több időbe telik, mint várnánk, hiszen a program az eljárást mindaddig ismétli, amíg az eredmény „tökéletes" nem lesz.

Ha valaki nem az eredeti operációs rendszert használja, vagy az újabb típusú lemezmeghajtóval rendelkezik, akkor a védelem esetleg nem működik. Tekintve, hogy a gyártó néhány módosítást végzett az operációs rendszerben a forgalmazás ideje alatt, azok a védelmi rendszerek, amelyek a kihasználják a rendszer pl. előbb említett sajátosságait, nem működnek helyesen.

4.7 Manipulációk a szinkronjellel

Ebben az alfejezetben a szinkronjel módosítására alapozott másolásvédelmi módszereket ismertetjük.

4.7.1 Gyilkos sávok

A kizárólagosan szinkronjellel teleírt sáv a másolóprogramra – éppúgy bármilyen más programra – hatással van. A sáv beolvasási kísérlete a programot tökéletesen leblokkolja, azaz nevéhez hűen: meggyilkolja.

Hogy hatásmechanizmusát tökéletesen megértsük, vissza kell emlékeznünk arra, hogy egy átlagos rutin milyen lépésekben olvas be egy byte-ot a lemezről.

- `CLV` a byte–ready vezérlés törlése.
- `CC BVC CC` várakozás a byte-ra.
- `LDA $1C01` a byte betöltése.

A rendszer olvasás előtt minden esetben várakozik a SYNC-byte-ra, de az a gyilkos sáv esetén „nem várat magára". Miután a vezérlő a szinkronjelet azonosította, az adatok olvasását hardveresen félbeszakítja, és folytatását csak a jel végén engedélyezi újra. Eközben a byte–ready vezetéket a szinkronjel foglalja le. Amint a következő programrész szemlélteti, ez a jelenség egyértelműen egy végtelen ciklus kialakulásához vezet. A másolóprogram, amely nincs felkészülve a gondosan előkészített sáv fogadására, ezen a ponton automatikusan „elvérzik".

Az igaz, hogy az ötlet már rég nem ismeretlen a zugmásolók körében, és a legtöbb másolóprogramnak nem jelent akadályt a gyilkos sáv feldolgozása. Az azonban, hogy a másolóprogram a gyilkos sávok minden lehetséges változatára felkészüljön, szinte reménytelen vállalkozás.

4.7.2 Megnyújtott szinkronjelek

A gyilkos sávok egyik, a védelem szempontjából ígéretes módosítása a szinkronjelek megnyújtása. A módosítás lényege az, hogy a sávot szinkronjelekkel töröljük, majd beleírunk néhány adatot. A lekérdező program pontosan tudja, hogy hol keresse az adatokat, és milyen hosszúak a szinkronjelek.

A másolóprogramok nem tudják megállapítani a szinkronjelek hosszát, és így ha sikerül is elérniük az adatokat, a másolat vagy a jelek hosszában, vagy az adatokban el fog térni az eredetitől. A végeredmény tehát az, hogy bármily egyszerű a védelem, a lemez „másolhatatlan".

A következő program – amely a $0527-es címről indul – létrehozza a védelmet.

```
10 OPEN1,8,15
20 READ HD$:IF VAL(HD$)=-1 THEN100
23 GOSUB 1000
26 PRINT N,HD$
30 SU=SU+D:PRINT#1,"M-W"CHR$(N)CHR$(5)CHR$(1)CHR$(D)
40 N=N+1:GOTO20
100 IFSU<>5576 THEN PRINT"HIBA AZ ADATOKBAN":STOP
110 INPUT"SAV";T
120 PRINT#1,"M-W"CHR$(40)CHR$(5)CHR$(1)CHR$(T)
130 PRINT#1,"M-E"CHR$(39)CHR$(5)
140 CLOSE1
200 END
400 DATA AD,00,1C:REM        LDA   1C00
410 DATA 29,9F   :REM        AND  #9F
420 DATA 09,08   :REM        ORA  #08
430 DATA 8D,00,1C:REM        STA   1C00
440 DATA 20,A3,FD:REM        JSR   FDA3
450 DATA A2,00   :REM        LDX  #00
460 DATA BD,23,05:REM        LDA   0523,X
470 DATA 50,FE   :REM        BVC   0512
480 DATA B8      :REM        CLV
490 DATA 8D,01,1C:REM        STA   1C01
500 DATA E8      :REM        INX
510 DATA E0,06   :REM        CPX  #06
520 DATA D0,F2   :REM        BNE   050F
530 DATA 20,00,FE:REM        JSR   FE00
540 DATA 4C,9E,FD:REM        JMP   FD9E
550 DATA 53,54,55,56
560 DATA A9,24   :REM        LDA  #24
570 DATA 85,0A   :REM        STA   0A
580 DATA A9,E0   :REM        LDA  #E0
590 DATA 85,02   :REM        STA   02
600 DATA A5,02   :REM        LDA   02
610 DATA 30,FC   :REM        BMI   052F
620 DATA 60      :REM        RTS
630 DATA -1      :REM  PROGRAM VEGE
1000 REM HEXABOL DEC.-BE
1010 H$="0123456789ABCDEF"
1020 IF LEN(HD$)=0 THEN PRINT"HIBA":STOP
1030 IF LEN(HD$)<> 2 THEN HD$=LEFT$(HD$,2)
1040 H=0
1050 FOR ZZ=1 TO 16
```

```
1060 IF LEFT$(HD$,1)=MID$(H$,ZZ,1) THEN H=ZZ:GOTO 1080
1070 NEXT ZZ
1080 H=H-1
1090 L=0
1100 FOR ZZ=1 TO 16
1110 IF RIGHT$(HD$,1)=MID$(H$,ZZ,1) THEN L=ZZ:GOTO 1130
1120 NEXT ZZ
1130 L=L-1
1140 D=H*16+L
1150 RETURN
READY.
```

A következő program a védelem lekérdezésére szolgál. Kezőcíme a lemeztárban $0600, indítási címe pedig $06A.

```
10 OPEN1,8,15,"I"
20 READ HD$:IF VAL(HD$)=-1 THEN100
23 GOSUB 1000
25 SU=SU+D
30 PRINT#1,"M-W"CHR$(N)CHR$(6)CHR$(1)CHR$(D)
32 PRINTN,HD$
40 N=N+1:GOTO20
100 IFSU<>9626 THEN PRINT"HIBA AZ ADATOKBAN":STOP
110 INPUT"SAV";T
120 PRINT#1,"M-W"CHR$(75)CHR$(6)CHR$(1)CHR$(T)
135 PRINT#1,"M-E"CHR$(74)CHR$(6)
140 FORN=1TO500:NEXT
145 PRINT#1,"M-R"CHR$(16)CHR$(0)CHR$(1)
150 GET#1,A$:A=ASC(A$+CHR$(0))
170 IFA=255THEN PRINT"A VEDELEM RENDBEN":END
180 PRINT"A VEDELEM HIBAS":END
400 DATA AD,00,1C:REM        LDA   1C00
410 DATA 29,9F   :REM        AND  #9F
420 DATA 09,08   :REM        ORA  #08
430 DATA 8D,00,1C:REM        STA   1C00
440 DATA AD,00,1C:REM        LDA   1C00
450 DATA 30,FB   :REM        BMI   060A
460 DATA B8      :REM        CLV
470 DATA AD,01,1C:REM        LDA   1C01
480 DATA A2,00   :REM        LDX  #00
490 DATA 50,FE   :REM        BVC   0615
500 DATA B8      :REM        CLV
510 DATA AD,01,1C:REM        LDA   1C01
520 DATA DD,46,06:REM        CMP   0646,X
530 DATA D0,22   :REM        BNE   0642
540 DATA E8      :REM        INX
550 DATA E0,05   :REM        CPX  #05
560 DATA D0,F0   :REM        BNE   0615
570 DATA AD,00,1C:REM        LDA   1C00
580 DATA 30,FB   :REM        BMI   0625
590 DATA A2,00   :REM        LDX  #00
600 DATA A0,00   :REM        LDY  #00
610 DATA E8      :REM        INX
620 DATA D0,01   :REM        BNE   0632
630 DATA C8      :REM        INY
640 DATA AD,00,1C:REM        LDA   1C00
650 DATA 10,F7   :REM        BPL   062E
660 DATA C0,38   :REM        CPY  #38
670 DATA 90,07   :REM        BCC   0642
```

```
680 DATA A9,FF    :REM        LDA  #FF
690 DATA 85,10    :REM        STA   10
700 DATA 4C,9E,FD:REM        JMP   FD9E
710 DATA A9,00    :REM        LDA  #00
720 DATA F0,F7    :REM        BEQ   0630
730 DATA 53,54,55,56
740 DATA A9,24    :REM        LDA  #24
750 DATA 85,0C    :REM        STA   0C
760 DATA A9,E0    :REM        LDA  #E0
770 DATA 85,03    :REM        STA   03
780 DATA A5,03    :REM        LDA   03
790 DATA 30,FC    :REM        BMI   0652
800 DATA 60       :REM        RTS
810 DATA -1,      :REM PROGRAM VEGE
1000 REM HEXABOL DEC.-BE
1010 H$="0123456789ABCDEF"
1020 IF LEN(HD$)=0 THEN PRINT"HIBA":STOP
1030 IF LEN(HD$)<> 2 THEN HD$=LEFT$(HD$,2)
1040 H=0
1050 FOR ZZ=1 TO 16
1060 IF LEFT$(HD$,1)=MID$(H$,ZZ,1) THEN H=ZZ:GOTO 1080
1070 NEXT ZZ
1080 H=H-1
1090 L=0
1100 FOR ZZ=1 TO 16
1110 IF RIGHT$(HD$,1)=MID$(H$,ZZ,1) THEN L=ZZ:GOTO 1130
1120 NEXT ZZ
1130 L=L-1
1140 D=H*16+L
1150 RETURN
READY.
```

A védelmet az 1-es sávra hoztuk létre. Azonban a védelem létrehozása előtt a lemezt az eredeti operációs rendszer formázórutinjával formázzuk meg!

5. A PROGRAMOK VÉDELME MÁGNESSZALAGON

A felhasználók többsége a C 64-es gépeken a programokat mágneslemezen, diszken tárolja. Ennek ellenére talán mégsem fölösleges áttekinteni a mágnesszalagok másolásának néhány védelmi módszerét is. Programjaink védelmén kívül jól használható ismereteket is szerezhetünk a C 64-es működéséről, „lelkéről". Szalagról szalagra másolni technikailag lényegesen egyszerűbb, mint mágneslemezről. Egy közönséges átjátszó zsinórral egyik magnóról a másikra másolható a teljes tartalom. Miért érdemes akkor a kazetták védelmével foglalkozni? Azért, mert már egyetlen ilyen közvetlen átjátszás is gyengébb minőségű másolatot eredményez; a másolat másolata még rosszabb minőségű lesz, és előbb-utóbb már használhatatlan lesz a program. Az ilyen „olcsó" trükknél nehézséget okoz a magnófej külön beállítása is.

Lemezes másolóprogramból nagyon sok létezik, ezekről már eddig is olvashattunk, többségük nagyon jó minőségű másolatot készít a védett programokról is. Mágnesszalaghoz lehetetlen ilyen színvonalú másolóprogramot készíteni. A legtöbb programozó, ill. forgalmazó ui. saját jelrendszerben dolgozik, így általános másolóprogram nem készíthető. Minden egyes cég termékéhez készíthető ugyan az adott formátumot kezelni képes másoló, de ez az eljárás nem az igazi. Nem is elég általános (pl. az NSZK-ban és nálunk a kazettamásolók többsége a Turbo-Tape formátumot használja).

5.1 Adatok, programok betöltése, tárolása BASIC-ben

A másolás elleni védelem ismertetése előtt érdemes egy kicsit foglalkozni a mágnesszalagra való programrögzítéssel és -betöltéssel. A C 64-es operációs rendszere két lehetőséget kínál erre:

LOAD–SAVE, ill. OPEN–CLOSE.

a) LOAD–SAVE utasításpár
A tárban lévő program szalagra írása:

```
SAVE "programnév"
SAVE "programnév",1
```

A szalagon tárolt program betöltése a tárba:

```
LOAD "programnév"
LOAD "programnév",1
```

A szalagon tárolt program ilyen esetben két részből áll. Az első rész a fejléc (angolul header), amely a program nevét, hosszát, tárbeli elhelyezését tartalmazza. A második rész maga a program. Ez a felosztás megfigyelhető a betöltés közben is, mivel a számítógép a fejlécet kétszer egymás után elolvassa, kiírja a program nevét, és csak ez után tölti be a BASIC programot.

b) OPEN...PRINT#...CLOSE
Adatokat – számokat, változók értékeit, betűket – rögzíthetünk és tárolhatunk mágnesszalagon a következő módon.

Az `OPEN 1,1,1,"file-név"` utasítással megnyitunk a szalagon egy adatfile-t, azaz tetszőleges adatokat írhatunk a szalagra. A `PRINT#1,...` utasítással írhatjuk az adatokat a szalagra. Hatása ugyanaz, mint a `PRINT...` utasításé, csak most a képernyő helyett a szalagra kerülnek az adatok.

A szalagkezelő utasítások sorát a `CLOSE 1,` utasítással kell lezárnunk. Ha ezt elmulasztjuk, vagy hiányos lesz az adathalmaz, vagy hibajelzést kapunk beolvasáskor.

Az OPEN, ill. a CLOSE utasításban szereplő paraméterek közül az első az ún. logikai file száma, értéke 0 és 255 közé eshet. Ez azért fontos, mert a PRINT# és a CLOSE utasításokban ugyanerre a számra kell hivatkozni. Így azonosítja az operációs rendszer az összetartozó három utasítást. A második paraméter az egységszám, amelyre a kiíratás irányul, kazettás egységnél ez 1. A harmadik paraméter a másodlagos cím, amelynek különböző funkciói lehetnek. Amennyiben mágnesszalagot használunk, és másodlagos címként 1-et adunk meg, az operációs rendszer ezt úgy azonosítja, hogy „adatok írására megnyitottunk egy adatfile-t". Amennyiben a szalagon lévő adatokat kívánjuk visszaolvasni, az adott adatfile-t 0 másodlagos címen kell megnyitnunk:

```
OPEN 1,1,0,"file-név"
```

A szalagon lévő adatok beolvasásához két utasítást használhatunk:

```
GET#1,... változók
```

vagy

```
INPUT#1,... változók
```

Az adatok beolvasását ebben az esetben is a CLOSE 1 utasítással kell lezárnunk, különben ugyanaz érvényes, mint a GET, ill. a PRINT utasításra, csak a billentyűzet helyett a mágnesszalagra vonatkoztatva.

Tekintsünk most néhány példát! A következő programmal megadott számú adatot, numerikus változót olvashatunk be a tárba a billentyűzetről, és ezeket szalagon tároljuk.

```
10 OPEN1,1,1,"ADATOK"
20 PRINT"ADATOK SZAMA"
30 INPUT A:PRINT#1,A
40 FORI=1 TO A
50 PRINTI
60 INPUTB:PRINT#1,B
70 NEXTI
80 CLOSE1

READY.
```

A szalagon rögzített számokat visszatölthetjük a tárba:

```
10 OPEN1,1,0,"ADATOK"
20 INPUT#1,A
30 FORI=1TOA
40 INPUT#1,B
50 PRINTB
60 NEXTI
70 CLOSE1

READY.
```

A szalagkezelő utasításokat igazán csak áttekintésnek szántuk.

Most rátérünk a másolás elleni védelem legalapvetőbb, legegyszerűbb módszereinek ismertetésére. Az egyik lehetőség az, hogy a szalagra néhány számot írunk a programunk után. Betöltés után ellenőrizhetjük, hogy megváltoztak-e ezek a számok, és csak akkor engedélyeztetjük a program futását, ha a számsor változatlan.

Legyenek pl. az ellenőrző számok: 1, 3, 2, 4, a visszakérdező program pl. a következő:

```
10 OPEN1,1,0,"ADATOK"
20 INPUT#1,A,B,C,D
30 CLOSE1
40 IFA<>1 ORB<>3ORC<>2ORD<>4 THEN STOP
50 PRINT"MASOLAS ENGEDELYEZVE"

READY.
```

Ez az egyszerű védelem már megakadályozza, hogy LOAD/SAVE paranccsal betölthető, ill. rögzíthető legyen a program. A betöltő program elkészítése azért jelent feladatot, mert a tárolt adatok számát nem ismerheti a „tolvaj". Mint tudjuk, az adatfile másolása a GET# utasításra épül, amelynek hatására

folyamatosan olvassa és tárolja az adatokat a gép, mindaddig, amíg az ST rendszerváltozó értéke nem jelzi az adathalmaz végét. Az utolsó byte betöltése után ST értéke 64 lesz, amit az operációs rendszer egy CHR$(0) karakterrel jelez. Az adatfile végét jelző utolsó byte-ot egy CHR$(0) karakter beírásával előállíthatjuk anélkül, hogy az összes adatot beolvastuk volna. Így az ST rendszerváltó értéke 64 lesz, pedig a program nem olvasta be a teljes ellenőrző adatsort. Így a lekérdező program automatikusan leállítja a programot.

Tekintsünk most példaként egy-egy másoló, védekező és ellenőrző programot:

```
10 DIMD$(1000)
20 INPUT"FILE NEVE";N$
30 OPEN 1,1,0,N$
40 I=0
50 GET#1,D$(I)
60 IF ST=0 THEN I=I+1:GOTO50
70 CLOSE1
80 INPUT"KAZETTACSERE";N$
90 OPEN1,1,1,N$
100 FOR J=0 TOI
110 PRINT#1,D$(J)
120 NEXTJ
130 CLOSE1

READY.


10 OPEN1,1,1,"FILE NEVE"
20 INPUT#1,"COMMO"
30 PRINT#1,CHR$(0)
40 PRINT#1,"DORE"
50 CLOSE1

READY.


10 OPEN1,1,1,"FILE NEVE"
20 INPUT#1,A$
30 GET#1,B$
40 INPUT#1,B$
50 CLOSE1
60 IF A$<>"COMMO" OR B$<>"DORE"THEN STOP
70 PRINT"MASOLAS ENGEDELYEZVE"

READY.
```

Nézzünk most egy másik lehetőséget a programvédelemre, még mindig BASIC-ben. Az operációs rendszer nemcsak a programokat, hanem az adatokat is két egységen tárolja a szalagon. Ebben az esetben a blokkfej a file nevét és adatait tartalmazza, a második rész pedig az adatokat, adatblokkokban. Betöltéskor a blokkok egyesével kerülnek először a kazettapufferbe, ahonnan a BASIC program továbbítja a tárba. A következő blokk beolvasása csak akkor indul, ha a kazettapuffer üres. Ezt figyelembe véve a másolóprogramban is elég, ha az adablokkok számát ismerjük, és nem fontos tudni a byte-ok számát. Így

kikerülhető a CHR$(0) kód vizsgálata, tehát az ellenőrző programunkat érdemes kicsit elrejteni, zavarossá tenni (lásd a 3. fejezetben).

Az OPEN utasításban megadott másodlagos címet is felhasználhatjuk a védelemre. Emlékezzünk vissza, hogy az utasításban szereplő harmadik paraméter jelöli a másodlagos címet.

Az OPEN 1,1,2,"file-név" utasítás hatására az operációs rendszer egy EOT jelet ír az adatok után a szalagra. Az EOT a szalag végét jelzi (End Of Tape), azaz megjelöli a szalagon azt a helyet, amely után már sem programot, sem adatot nem találunk. Amikor ezt a jelet beolvassa az operációs rendszer, megáll és hibaüzenetet ad: Device not present error. Ezért programból nem tudjuk ellenőrizni az EOT jelet, hiszen a gép leáll és alaphelyzetbe kerül.

A másodlagos címben rejlő védelmi lehetőségek kihasználásához nézzük meg, hogyan helyezkednek el programjaink a tárban. A BASIC kezdőcím a 43-as és a 44-es tárcímeken található, alsó byte, felső byte alakban, és értéke alaphelyzetben 2049 ($0801). BASIC programunkat azonban erről a tárterületről áthelyezhetjük. Így ha valaki futtatni akarja a szokásos 2049-es címtől, nem fog működni.

Ha a BASIC kezdőcímet pl. 3000-re akarjuk módosítani, a következőképpen járhatunk el. A 3000-et átírjuk hexadecimális alakba: 3000 = = 11*16*16+11*16+8 = $0BB8. A 43-as címre beírjuk – POKE utasítással – az alsó byte értékét (B8), a 44-es címre pedig a felső byte értékét (0B), természetesen decimális alakban. Az „új" BASIC kezdőcím elé, 2999-re pedig 0-t kell írnunk.

```
POKE 43,184: POKE 44,11: POKE 2999,0: NEW
```

Amennyiben egy „normál", azaz 2049 kezdőcímű BASIC program rögzítése előtt az előbbi áthelyezést végrehajtjuk, a következő betöltéskor a kezdőcím értéke már 3000 lesz. Így ha a LOAD parancsban a másodlagos címnek 1-et adunk meg: LOAD "programnév",1,1 formában, akkor a program nem futtatható, mivel ez a parancs az eredeti helyére tölti vissza a programot. Tehát egy viszonylag egyszerű, persze éppen ezért könnyen kiismerhető védelem, ha a BASIC kezdőcímet eltoljuk, és másoláskor úgy adjuk meg a másodlagos címet a SAVE parancsban, hogy ne tiltsa le az eltolást.

Végül térjünk még ki arra, hogyan írhatjuk a szalagra a programunk után az EOT jelet, közvetlenül a SAVE parancsból. Ezt vagy a 2-es, vagy pedig a 3-as másodlagos címmel érhetjük el.

A BASIC-ből kiadható szalagvezérlő utasításokban tehát a másodlagos címek jelentése a következő:

OPEN 1,1,0,"név" :	adatfile megnyitása beolvasásra
OPEN 1,1,1,"név" :	adatfile megnyitása kiíratásra
OPEN 1,1,2,"név" :	adatfile megnyitása kiíratásra, a végére EOT jelet írva
SAVE"név",1 :	program tárolása, eltolási lehetőséggel
SAVE"név",1,1 :	program tárolása, eltolási lehetőség nélkül

SAVE"név",1,2 :	program tárolása eltolási lehetőséggel, végére EOT jelet írva
SAVE"név",1,3 :	program tárolása, eltolási lehetőség nélkül, végére EOT jelet írva
LOAD"név",1 :	program betöltése tetszőleges BASIC kezdőcímre
LOAD"név",1,1 :	program betöltése az eredeti BASIC kezdőcímre

5.2 Betöltés és tárolás gépi kódú programokból

Programjainkat egy fokkal jobban el tudjuk rejteni az illetéktelen szemek elől, ha vezérlő utasításainkat a BASIC utasítások gépi kódú megfelelőire építjük. Tekintsük tehát át a SAVE, LOAD, OPEN, GET, PRINT, CLOSE parancsok gépi kódú változatait és használatukat.

Az OPEN a SAVE és a LOAD parancsokban általában három paramétert kell megadni, a file számát, az egység számát és a másodlagos címet. Ezek értékét megadhatjuk két rendszerrutinnal, a FILPAR és a FILNAM rutinokkal.

A FILPAR kezdőcíme $FFBA (65466), végrehajtásához a file számát az akkumulátorba, az egység számát az X regiszterbe, a másodlagos címet az Y regiszterbe kell tölteni.

A FILNAM kezdőcíme $FFBD (65469), a hivatkozáshoz az akkumulátorban kell lenni a file-név hosszának, az X/Y regiszterben pedig a file-név kezdőcímének, alsó/felső byte alakban.

Nézzünk egy példát egy olyan gépi kódú programra, amely elvégzi az értékadást!

A file-név kezdőcíme: $C100
A program kezdőcíme: $C000
A file neve: PROGRAM

A paraméterek megadása:

```
C000 LDA #$01    file száma
C002 LDX #$01    egység száma
C004 LDY #$01    másodlagos cím
C006 JSR $FFBA   FILPAR rutin
C009 LDA #$08    file-név hossza
C00B LDX #$00    alsó byte
C00D LDY #$C1    felső byte
C00F JSR $FFBD   FILNAM rutin
```

Az egyes parancsokhoz tartozó kezdőcímek a következők:

SAVE: $FFD8 (65496)
LOAD: $FFD5 (65493)
OPEN: $FFC0 (65472)
CLOSE: $FFC3 (65475)

SAVE: Végrehajtás előtt ki kell jelölni azt a tárterületet, amelyet szalagra akarunk másolni. A terület kezdőcímét a nulláslap két byte-jára, a végcímnél eggyel nagyobb értéket pedig az X, Y regiszterbe kell beírni.

Készítsünk egy betöltőrutint, amellyel az $1000 (4096) és a $2000 (8192) címek közé eső tárterületet másolhatjuk a szalagra. A file-paraméterek megadása után a nulláslapon használhatjuk az $FB (251) és az $FC (252) címeket.

```
C012 LDA #$00    alsó byte (kezdőcím)
C014 STA  $FB
C016 LDA #$10    felső byte (kezdőcím)
C018 STA  $FC
C01A LDA #$FB
C01C LDX #$01    alsó byte (végcím+1)
C01E LDY #$20    felső byte (végcím+1)
C020 JSR $FFD8   SAVE rutin
```

Megjegyezzük, hogy az így tárolt programot is betölthetjük BASIC-ből, LOAD paranccsal.

LOAD (gépi kódban): A LOAD rutin hívása előtt az akkumulátorba nullát kell tennünk. Ha ui. az akkumulátor tartalma nem nulla, az operációs rendszer LOAD helyett VERIFY-t (ellenőrzést) hajt végre. Ha a másodlagos cím 1 volt a tároláskor, tehát a BASIC területet nem változtattuk meg, a beolvasó rutint a következő rövid gépi kódú programmal hívhatjuk meg:

```
C012 LDA #$00    választás:LOAD vagy VERYFY
C014 JSR $FFD5   LOAD rutin
```

Ha a SAVE-ben a másodlagos cím 0 volt, akkor a kezdőcímet meg kell adnunk, mégpedig az X, Y regiszterben, alsó byte/felső byte alakban. Legyen pl. a kezdőcím $1989, a beolvasó rutin a következő:

```
C012 LDA #$00    választás:LOAD vagy VERIFY
C014 LDX #$89    alsó byte (kezdőcím)
C016 LDY #$19    felső byte (kezdőcím)
C018 JSR $FFD5   LOAD rutin
```

Ha a betöltés megszakad, a LOAD rutin az átviteli (carry) bitet 1-re állítja. Amennyiben a szalagon egy EOT jelet talál, az akkumulátor tartalmát 5-re állítja. Az EOT vizsgálata éppen ezért nagyon egyszerű gépi kódból, védelemre mégsem érdemes alkalmazni, hiszen feltörése is hasonlóan egyszerű. Olvasáskor

az adathibát az ST státuszváltozó értéke mutatja, címe $90 (144). Az adatolvasás abban az esetben hibátlan, ha ST értéke nulla.

OPEN: Az adatcsatorna megnyitása előtt a FILPAR és a FILNAM rutinokat hívjuk a paraméterek rögzítésére, majd ugrás az OPEN rutin kezdőcímére $FFC0 (65472). Ezután betöltjük a file logikai számát az X regiszterbe, és meghívjuk a CKOUT rendszerrutint, az $FFC9 (65481) címen. Most jöhet a BASOUT rutin hívása az $FFD2 (65490) címen úgy, hogy előtte a soron következő byte-ot mindig betesszük az akkumulátorba. A tényleges adatfelírást tehát a BASOUT rutin végzi, amely a PRINT# parancs megfelelője.

A következő programmal egy BASIC programot rögzíthetünk a tárból mágnesszalagra, adatfile-ként. Programunk kezdőcíme a $2B (43), $2C (44) tárcímeken, végcíme pedig a $2D (45), $2E (46) tárcímeken legyen, alsó byte/felső byte alakban. A ciklusszámlálást az $FB (251) és az $FC (252) tárcímeken végezzük.

```
C000 LDA #$01
C002 LDX #$01
C004 LDY #$01
C006 JSR $FFBA
C009 LDA #$00
C00B JSR $FFBD
C00E JSR $FFC0
C011 LDX #$01
C013 JSR $FFC9
C016 SEC
C017 LDA $2D
C019 SBC $2B
C01B PHP
C01C JSR $FFD2
C01F PLP
C020 LDA $2E
C022 SBC $2C
C024 JSR $FFD2
C027 LDA $2B
C029 STA $FB
C02B LDA $2C
C02D STA $FC
C02F LDY #$00
C031 LDA $FB
C033 CMP $2D
C035 BNE $C03D
C037 LDA $FC
C039 CMP $2E
C03B BEQ $C04A
C03D LDA ($FB),Y
C03F JSR $FFD2
C042 INC $FB
C044 BNE $C031
C046 INC $FC
C048 BNE $C031
C04A LDA #$01
C04C JSR $FFC3
C04F JMP $FFCC
```

A program visszaolvasása a szalagról a tárba hasonló módon lehetséges. Most a CKOUT rutin helyett a CHKIN rutint hívjuk, amelynek kezdőcíme $FFC6 (65478). A file tartalmát byte-onként BASIN rutinnal olvashatjuk be, kezdőcíme $FFCF (65487). Ez BASIC-ben a GET# utasításnak felel meg. Vigyázzunk, mert a BASIN rutin végrehajtása közben az Y regiszter tartalma változik, ezt mindig vissza kell állítani.

```
C000 LDA #$01
C002 LDX #$01
C004 LDY #$00
C006 JSR $FFBA
C009 LDA #$00
C00B JSR $FFBD
C00E JSR $FFC0
C011 LDX #$01
C013 JSR $FFC6
C016 JSR $FFCF
C019 STA $FB
C01B JSR $FFCF
C01E STA $FFCF
C01E STA $FC
C020 LDA $2B
C022 STA $2D
C024 LDA $2C
C026 STA $2E
C028 JSR $FFCF
C02B LDY #$00
C02D STA ($2D),Y
C02F INC $2D
C031 BNE $C035
C033 INC $2E
C035 DEC $FB
C037 LDA $FB
C039 CMP #$FF
C03B BNE $C03F
C03D DEC $FC
C03F CMP  #$00
C041 BNE $C028
C043 LDA $FC
C045 BNE $C028
C047 LDA #$01
C049 JSR $FFC3
C04C JSR $FFCC
C04F JSR $A659
C052 JSR $A533
C055 JMP $A474
```

CLOSE: Az előző két programban láthatjuk a CLOSE rutin meghívását. Természetesen az írás/olvasás művelet után az adatcsatornát a BASIC-hez hasonlóan gépi kódban is le kell zárni, amelynek címe $FFC3 (65475). A lezárás előtt a file logikai számát be kell írnunk az akkumulátorba, utána pedig a CLRCH rutint kell meghívni, az $FFCC (65484) címen. Ezzel a rutinnal visszaállítjuk az input/output egységet alapértelmezésre, azaz billentyűzet- és képernyőhasználatra.

Az ilyen módon, azaz adatfile-ként tárolt programjaink védelmére felhasználhatjuk, hogy a BASIC program jó néhány CHR$(0) karaktert tartalmaz, így a file végét nehezen lehet megtalálni. Összegezve tehát az elmondottakat, a következőket tehetjük programjaink védelmére. A BASIC programot az elmondottak szerint adatfile-ként tároljuk úgy, hogy mögé egy oda nem illő számsort is fölírunk a szalagra. A gépi kódú betöltőrutin felismeri, hogy mi az, ami nem tartozik a programhoz, és átugorja az ilyen adatokat, a másolóprogram azonban nem. Így akár annyi fölösleges adatot is beolvashat a szalagról, hogy azok már el sem férnek a tárban, és így a másolás félbeszakad. Mindez persze a nagy hátránya is ennek a módszernek, hiszen az adatfile hossza a betöltési időt is nagyon megnöveli.

Mielőtt továbbmennénk, az adatok tárolásával kapcsolatban szeretnénk felhívni a figyelmet arra, hogy a RAM-ban az $A111 címtől $BFFF-ig és az $D000-címtől $FFFF-ig nem tárolhatunk programot, a második sávban még gépi kódú programot sem. Erre azért érdemes figyelnünk, mert az itt elhelyezett adatok bizonyos fokig védettek a másolással szemben.

Így pl., ha a programunk 38 Kbyte-nál hosszabb és az $A000 címen túl is folytatódik, akkor a SAVE előtt meg kell változtatni a tárfelosztást. Ezt megtehetjük a következő assembler utasításokkal:

```
LDA#$36
STA#$01
```

Így a BASIC interpreter átkerül az $A000 címről az alatta lévő RAM területre. Ezért mielőtt BASIC-be visszatérnénk, az $01-es cím tartalmát állítsuk vissza #$37-re:

```
LDA#$37
STA#$01
```

Ezt a témát befejezve, a következő részben arról lesz szó, hogyan lehet a tár tartalmát $E000-tól $FFFF-ig kimenteni és ugyanoda visszatölteni.

5.3 Autostart mint a másolás egyik ellenszere

5.3.1 A LOAD/SAVE rutinok ismertetése

A másolás elleni védekezéshez nem feltétlenül szükséges olyan rutinokat írni, amelyek elvégzik a betöltést és a kimentést. Nyugodtan felhasználhatjuk az operációs rendszer ezt végző programrészleteit csekély módosítással. Ehhez

feltétlenül szükség van egy monitorprogramra. Alapvetően két lehetőség kínálkozik. Az egyik, hogy a LOAD/SAVE rutinokra mutató vektorokat átirányítjuk a RAM-ba egy általunk írt programra. Mivel azonban e programok megírása meglehetősen bonyolult, erre most nem térünk ki részletesen.

A másik megoldás sokkal egyszerűbb. A nekünk kellő ROM részletet átkapcsoljuk a RAM-ra, azt ezután igény szerint módosíthatjuk. Monitorprogrammal az átkapcsolás nagyon egyszerű:

```
.T E000 FFFF E000
```

Elképzelhető, hogy a monitorprogramban, amivel a felhasználó rendelkezik, nem ez a megfelelő utasítás. Ebben az esetben a kézikönyvben kell utánanézni az ennek megfelelő parancsnak. Ha az alkalmazott program az operációs rendszert csak a BASIC interpreterrel együtt tudja átkapcsolni, akkor hajtsuk végre a következő parancsot is:

```
.T 000 BFFF A000
```

Most írjuk be a következőt:

```
.M 0001
```

A parancs hatására megjelenik a képernyőn 8 darab hexadecimális szám, amelyek közül az első $37. Ezt változtassuk $35-re. Ezzel megtettük az előkészületeket. Először töltsük be a programot a memóriába a $E000-as címtől kezdve. Ezután tároljuk a tár tetszőleges helyén, pl. $1200-tól kezdve. Tételezzük fel, hogy a program hossza $500 byte. A tároláshoz ezek szerint a $E000-tól a $E500-ig terjedő rész kell. Ezután módosítsuk a betöltőrutint a monitorprogrammal. Ha ezek után a $1200-tól $1700-ig terjedő részt tároljuk külső egységen (kazetta), az a visszatöltődéskor a $E000-tól kezdődő címre kerül. Mielőtt továbbhaladnánk, a SAVE rutint vissza kell állítanunk, mégpedig a ROM visszakapcsolásával. Ehhez az egyes tárcímbe $37-et kell írni. Mivel az operációs rendszert újra megváltoztattuk, célszerű a ROM-ot a RAM-ba újra átmásolni.

Az operációs rendszer módosításával meg lehet változtatni a tárolás sebességét, ill. formátumát. Ilyen változtatásokhoz azonban a ROM rutinjainak – amelyek igen bonyolultak – alapos ismerete szükséges. Ennél az is jobb, ha teljesen új töltő- és tárolóprogramot írunk céljainkra; ez azonban könyvünknek nem témája. Aki az operációs rendszer belső rutinjaival részletesebben kíván foglalkozni, annak a *A C 64-es belső felépítése* című könyvet tudjuk ajánlani.

5.3.2 Önindító programok

Egy kis intermezzó után kanyarodjunk vissza ismét fő témánkhoz. Lemezen az önindítás önmagában nem jelent túl nagy védelmet, kazettánál viszont más a helyzet. Itt nyugodtan mondható, hogy a kazettán tárolt önindító programok

másolására nincs lehetőség. Ezzel szemben kazettán jóval kevesebb lehetőség van az autostart megvalósítására, mint lemezen. Az egyik megoldást be is mutatjuk. Ehhez ismerni kell a gép működését, hogy a betöltés valójában hogyan történik.

A betöltés után az operációs rendszer meghatározott részére adódik át a vezérlés, méghozzá meghatározott vektorok által. Ha ezeket a vektorokat átállítjuk úgy, hogy az általunk írt program elejére mutassanak, akkor programunk magától fog indulni. A módosított vektort programunkkal együtt kell tárolni.

Fontos, hogy a program kimentésénél a másodlagos cím 1 legyen, mert különben a BASIC elejére, tehát $0801-re töltődne. A betöltés után a programnak vissza kell állítani a vektorokat a kezdeti értékre. Ezen kívül ésszerű a STOP és a STOP/RESTORE gombok letiltása. Enélkül ui. a program futása megszakítható és ezáltal másolható; ekkor pedig az autostart elvesztené ezt a funkcióját.

Melyek azok a vektorok, amelyek ezt megvalósíthatják? A 24. táblázatban felsoroljuk a lényeges vektorokat.

24. táblázat

Vektor	Eredeti érték	Funkció
$0302/$0303	$A483	BASIC sor bevitele
$0324/$0325	$F157	INPUT
$0326/$0327	$F1CA	PRINT
$0328/$0329	$F6ED	STOP

A STOP letiltásához a vektort a $F6E1 címre kell irányítani. Arra különösen kell figyelni, hogy a megszakítás vektorát (IRQ $0314/$0315) ne változtassuk meg, mert ez zavarná a LOAD és a SAVE rutint.

A $0801 címen kezdődő BASIC programokhoz a $0324/$0325 vektorokat kell használni; a $02A7-től a $02FF-ig lévő programokhoz pedig a $0302/$0303 vektorokat. Nem elhanyagolható, hogy a gép a képernyőtárat is el fogja menteni, ez azonban semmiféle fennakadást nem okoz. Példaprogramunk szempontjából fontos az is, hogy a BASIC program végcíme a $2D/$2E (45/46) címeken található. Hogyha programunk nem gépi kódú, akkor nekünk kell meghívni a RUN parancsot egy kis assembler rutinnal (gépi kódú programok esetében nem kell a RUN parancsot meghívni).

Végül a vektorok inicializálására két rutin:

$E453: $0300-tól a $0303-ig lévő vektorokat,

$FF84: $0314-től a $0333-ig lévő vektorokat

hozza alapállapotba. Ezeket azonban nem célszerű használni, mert ez a STOP vektort is visszaállítaná.

Lássunk egy példaprogramot, amely bemutatja, hogyan lehet egy BASIC programot automatikusan elindítani.

```
100 REM**********************************
110 REM*** BASIC PROGRAM ELMENTESE ***
120 REM***        AUTOSTARTTAL        ***
130 REM***    GEPI KODU PROGRAM       ***
140 REM***     BASIC BETOLTOJE        ***
150 REM**********************************
160 REM**********************************
170 REM*** A PROGRAM HEXADECIMALIS ***
180 REM*** SZAMOKBAN VAN  TAROLVA  ***
190 REM***       EZT ATALAKITJA       ***
200 REM***     DECIMALIS SZAMMA       ***
210 REM*** S ELHELYEZI A MEMORIABA ***
220 REM*** ELLENORZI AZ EGYES DATA ***
230 REM***          SOROKAT           ***
240 REM**********************************
250 FORI=1TO80
270 READ A$:A$=LEFT$(A$,2)
280 B$=RIGHT$(A$,1)
290 A=ASC(A$)-48:IFA>9THENA=A-7
300 B=ASC(B$)-48:IFB>9THENB=B-7
310 A=A*16+B:L=L+A
312 POKE49151+I,A:L=L+A
319 NEXTI
320 IFL<>20610THENPRINT"HIBA VAN A BEUTESBEN":END
321 END
330 DATA A2,12     :REM LDX #$12
340 DATA BD,43,C0  :REM LDA $C043,X
350 DATA 9D,E8,07  :REM STA $07E8,X
360 DATA CA        :REM DEX
370 DATA 10,F7     :REM BPL $C002
380 DATA 20,D4,E1  :REM JSR $E1D4
390 DATA A9,01     :REM LDA #$01
400 DATA AA        :REM TAX
410 DATA A8        :REM TAY
420 DATA 20,BA,FF  :REM JSR $FFBA
430 DATA A9,93     :REM LDA #$93
440 DATA 20,D2,FF  :REM JSR $FFD2
450 DATA A9,E8     :REM LDA #$E8
460 DATA 8D,24,03  :REM STA $0324
470 DATA A9,07     :REM LDA #$07
480 DATA 8D,25,03  :REM STA $0325
490 DATA A9,E1     :REM LDA #$E1
500 DATA 8D,28,03  :REM STA $0328
510 DATA A9,36     :REM LDA #$36
520 DATA 85,01     :REM STA $01
530 DATA A9,24     :REM LDA #$24
540 DATA 85,FB     :REM STA $FB
550 DATA A9,03     :REM LDA #$03
560 DATA 85,FC     :REM STA $FC
570 DATA A9,FB     :REM LDA #$FB
580 DATA A6,2D     :REM LDX $2D
590 DATA A4,2E     :REM LDY $2E
600 DATA 20,D8,FF  :REM JSR $FFD8
610 DATA E6,01     :REM INC $01
620 DATA 4C,8A,FF  :REM JMP $FF8A
630 DATA A9,F1     :REM LDA #$F1
```

```
640 DATA 8D,25,03 :REM STA $0325
650 DATA A9,00    :REM LDA #$00
660 DATA 20,71,A8 :REM JSR $A871
670 DATA 4C,AE,A7 :REM JMP $A7AE

READY.
```

A programot SYS49152 "file-név" utasítással lehet indítani. Van azonban más lehetőség is. A programot el kell tolni az eredeti vektorokkal együtt. Ezután módosítjuk az operációs rendszert.

Ezeknek a módszereknek viszont hátrányai is vannak. Hiszen az elv visszafelé is működik, tehát lehetséges olyan betöltőprogram készítése, amely a vektorokat rögtön a betöltés után alaphelyzetbe hozza. Teljes védelmet tehát nem nyújt, de elég biztonságosnak mondható.

5.4 A kazettapuffer

A kazettapuffer funkciója speciális. Alapvetően kettős feladatot lát el: egyrészt ez a terület az adatok átmeneti helye – akár írni, akár olvasni akarunk; másrészt a program betöltésekor itt található a fejléc (header). Ez a következőképpen néz ki:

$033C	a fejléc típusa
$033D/$033E	kezdőcím
$033F/$0340	végcím
$0341–$0350	a program neve (kijelzett)
$0351–$03FF	az esetleges speciális adatok

A puffer helye a tárban alaphelyzetben $033C (828), de bármely szabad területre áthelyezhető. Vektora a $B2/$B3 címen található. Csupán ennek módosítása azonban nem elég a programvédelem szempontjából.

Mint már említettük, a puffer első byte-ja jellemzi a file típusát. A különböző értékek jelentése:

1: A file eltolható. Ekkor, ha a másodlagos cím 0, a file a kezelő által megadott értékre töltődik. Alaphelyzetben $0801-re, ami a BASIC tár kezdete.
2: Nincs fejléc, ebből következően adatblokkról van szó. Nincs cím és név.
3: A file nem eltolható. Betöltési helyét az adott cím határozza meg.
4: Adatfile fejléce.
5: EOT jel (file vége).

Az ezután következő négy byte a program kezdő- és végcímét tartalmazza. Ezt ki is lehet próbálni. Visszatöltéskor, amikor megjelenik a FOUND üzenet, állítsuk meg a programot a STOP billentyűvel.

Ezután a PEEK utasítást kell használni.

A kezdőcímhez: `?peek(829)+256*peek(830)`
A végcímhez: `?peek(831)+256*peek(832)`

A további byte-ok a file nevét tartalmazzák. A file neve – mint tudjuk – 16 karakter lehet. De miért, hiszen a pufferban bőven van hely?! Nos, a magyarázat igen egyszerű. A név hosszúsága ui. tetszőleges, kimentésnél semmi sem korlátozza. Betöltéskor az egész be is töltődik a tárba, mi azonban ebből csak 16 karaktert láthatunk. Erről könnyen meg is győződhetünk. Mentsünk ki egy programot 16 karakternél hosszabb névvel. Pl.:

```
SAVE "ez egy igen hosszú név"
```

Visszatápláláskor ezt fogjuk kapni:

```
FOUND ez egy igen hossz
```

Most állítsuk meg a programot a STOP gombbal.

Ezután a PEEK utasítással olvassuk vissza a puffer tartalmát. A teljes név benne lesz...

Ez az apróság igen sokrétűen felhasználható. Hiszen a beavatatlan nem tudhatja a file-név maradék részét, mi azonban – akik a programot írtuk – egy kis betöltő segítségével könnyen ellenőrizhetjük. Sőt, ezen akár tovább is mehetünk. A kazettappufferben ui. elrejthetünk e módon egy betöltőprogramot. Ennek a védelmen kívüli előnye az, hogy nem foglal helyet a tárban a betöltendő program előtt.

Írjuk meg az elrejtendő programot, és tegyük a név 16. karaktere mögé, azaz kezdőcíme legyen $0351. Ha a név rövidebb 16 karakternél, akkor pótoljuk szóközökkel. A tároláshoz az operációs rendszer rutinjait használjuk: SETNAM, SETFLS és a SAVE rutint (ld. 2.2 alfejezet).

A file-név legyen jó hosszú, hogy a helyére elférjen a betöltőrutin. Itt tulajdonképpen azt használjuk ki, hogy a gép – ahogy már az előbbiekben is írtuk – mentésnél nem korlátozza a file nevét. A maradék helyet töltsük ki nullával ($9D). Ezzel elejét vesszük a hibaüzeneteknek. Mielőtt a felhasználó a gyakorlati megvalósításhoz kezdene, célszerű először tanulmányoznia az 5.3 alfejezetet.

Most pedig nézzük az adatfile-okat. Azt már tudjuk, hogy az adatok mögé nem szükséges egy file-vég jelet beírni. Ismerkedjünk meg először a kiírás mechanizmusával. A formátumot a puffer $038C-től a $03FB-ig terjedő része határozza meg. Az írás is furcsán történik: ha a szalagra PRINT# utasítással adatot írunk, az nem lesz rögtön kiírva, hanem először a pufferbe kerül. Szalagra írás akkor történik, ha CLOSE utasítást kapott a rendszer, vagy a puffer betelt. Ekkor azonban a teljes puffer kiürül. Akkor is, ha mi eközben POKE utasítással a puffer tartalmát módosítottuk. A megoldás tehát egyszerű:

```
10 open2,1,1
30 print#2,"adat"
40 poke1010,230
50 close2
```

A visszaolvasás hasonlóan igen egyszerű:

```
10 open2,1,0
20 a=peek(1010)
30 ...
40 ...
```

Az A változó tartalma 230. Ezt kitalálni, akármilyen elszánt is a kódtörő, igen nehéz feladat.

5.5 A tárolás saját megformálása

5.5.1 A szalagműveletek vezérlése

Az igazán hatásos és feltörhetetlen védelem: saját magunk által kidolgozott formátum és betöltő program megírása. Ennek feltörése gyakorlatilag lehetetlen, hacsak nem használnak két magnót.
Ekkor viszont igen nagy a meghibásodás lehetősége. Saját betöltő programunknak előnye lehet az is, hogy meg tudjuk növelni a sebességet, s ez hosszabb programok esetén nem elhanyagolható. E rutinok továbbfejlesztése is jóval egyszerűbb, mint az eredeti rendszerrutinoké. Itt csak néhány ötletet adunk, a többit a felhasználó fantáziájára bízzuk.

Hogyan történik az adatok szalagra írása, egyetlen bit felírása? Ha valaki ezt tudja, az tulajdonképpen mindent tud, csak próbálni kell. A programot bitekre szedjük szét, majd olvasáskor ezt összerakjuk. De hogyan valósul meg a felírás? Az adatforgalom egy vezetéken bonyolódik le, amelyen kétféle jel lehet: magas és alacsony. Ha azonban egymás után küldenénk el a biteket, az egyformákat nem tudná a gép érzékelni. Ezért – mivel a különbségeket viszont jól érzi – mindig magasról alacsonyra kell váltani (lefutó él), és a kettő közt eltelt idő különbségét kell mérni. A rövidebb időköz legyen a 0 jel, a hosszabb pedig az 1.

Ezután már csak az idő mérése megoldandó feladat. Az időmérésre két önálló chip is van a C 64-esben. Céljaink megvalósítására nekünk elég az egyik, a CIA chip. Ennek kezdőcíme: $DC00 (56320).

Fontos regiszterei:

$DC04 (56324): az *A* timer alsö byte-ja
$DC05 (56325): az A timer felső byte-ja
$DC06 (56326): a *B* timer alsó byte-ja
$DC07 (56327): a *B* timer felső byte-ja
$DC0D (56333): megszakítást vezérlő regiszter
$DC0E (56334): az *A* timer vezérlőregisztere
$DC0F (56335): a *B* timer vezérlőregisztere

A chip, mint az előbbiekből kitűnik, két 16 bites órát tartalmaz. Ezek visszafelé számolnak, és nullánál megszakítást váltanak ki. Mi csak a *B* timert használhatjuk, mert a másik kell az operációs rendszernek.

A timer két részből áll: számlálóból és tárolórekeszből. Amit a tárolóba írunk, azonnal átkerül a számlálórészbe. Ott háromféle dolog történhet: a számlálót indítjuk, vagy a számlálót megállítjuk, vagy az indulóértéket rögzítjük. Azt, hogy e három eset közül melyik következzen be, a vezérlőrekesz tartalma dönti el. A vezérlőregiszterrel valósíthatjuk meg az 1-gyel való csökkentést is. Ez az ütem megegyezik a rendszer belső ütemével. Sőt, akár össze is köthetjük a két órát, ezáltal 32 bites számlálót kapva.

Ezek után lássuk, mit kell tudnunk a vezérlőregiszterről!

Az *A* vezérlőregiszter címe: $DC0E.
0. bit: 1 = *A* timer-start, 0 = *A* timer-stop.
1–2. bit számunkra érdektelen.
3. bit: 1 = 0-ig számlál, majd az indulóérték ismét a számlálóba kerül. Ezután leáll.
0 = 0-ig számlál, és a folyamat ismétlődik.
4. bit: 1 = az indulóérték a számlálóba kerül, függetlenül attól, hogy a számláló éppen mit csinál.
5. bit: 1 = a számláló külső jelforrással dolgozik.
0 = belső ütemmel dolgozik.
6–7. bit: nem érdekes.

A *B* vezérlőregiszter címe: $DC0F.
0–4. bitek: ua., csak a *B* timerre vonatkoztatva.
5–6. bitek: 00 = a timer belső rendszerütemet számol,
01 = külső jelforrást számol,
10 = akkor lép a *B*, ha az *A* lefutott,
11 = akkor lép a *B*, ha *A* lefutott és külső jelet is kap.
7. bit: érdektelen.

A megszakítóregiszter egyik feladata, hogy eldöntse, a timer lefutása váltson-e ki megszakítást, vagy sem (IRQ). A megszakításvezérlő másik funkciója annak figyelése, hogy egy bizonyos esemény bekövetkezett-e már, vagy sem.

A megszakítást vezérlő regiszter címe: $DC0D.
0. bit: 1 = az *A* timer lefutott.
1. bit: 1 = a *B* timer lefutott.
2–3. bit: érdektelen.
4. bit: 1 = a rendszer egy jelet felírt a szalagra.
5–6. bit: mindig 0.
7. bit: 1 = az események egyike megszakítást vált ki.

Némi magyarázatra szorul, hogyan lehet beállítani, hogy mely esemény idézzen elő megszakítást. Ha a 7. bitbe egyest írunk, ott lesz a megszakítás, ahol a vezérlőregiszterben is egyes van. Hasonló a módszer a törlésre is: a 7. bitbe 0-t írunk. Ahányadik bitbe ezután 1-est írunk, az a sorszámú esemény törlődik. Pl.:

```
lda #$84
sta $DC0D
```

Mivel $84 = %10000010, az első bitet állítják be, a többi változatlan marad.

A magasra állított bit engedélyezi, hogy az adott funkció megszakítást váltson ki. De arra, hogy az esemény bekövetkezett, a megszakításvezérlő megfelelő bitjéből következtethetünk. Feltétlenül tudnunk kell, hogy a bitek leolvasás után törlődnek. Ezért a megszakítási rutinunkban erre figyelni kell.

Ezek ismeretében elvileg már tudunk szalagra írni. Még azt kell tisztáznunk, hogyan lehet az adatokat felírni a szalagra. Ez a felírás egy vezetéken keresztül történik, amelyen különböző jelek küldhetők. Ilyen meghatározott jel pl. ami elindítja és megállítja a motort; vagy az, hogy a gép alacsony vagy magas jelet írjon fel a szalagra. Ezt a vezetéket mi a processzorporton keresztül tudjuk irányítani, amelynek elérési címe $01.

Az egyes bitek jelentése:
3. bit: a kazetta írásvonala
4. bit: 0 = le van nyomva a kazettán egy billentyű,
1 = nincs lenyomva billentyű
5. bit: 0 = motor be van kapcsolva,
1 = a motor nincs bekapcsolva

Mivel itt tulajdonképpen időmérés folyik, meg kell tiltani, hogy a processzor működése közben megszakítás lépjen fel, nehogy zavar keletkezzen. Ezért célszerű a programot a SEI utasítással kezdeni. Vigyázni kell arra is, hogy a videovezérlő, amely a monitoron a képet frissíti, ne okozzon megszakítást – célszerű a képernyőt kikapcsolni. Ez a $D011-es cím 4. bitjének törlésével érhető el. (Bár betöltéskor a videovezérlő időzítését is felhasználhatjuk!)

```
LDA $D011
AND #$EE
STA $D011
```

A visszakapcsolás a következőképpen érhető el:

```
LDA $D011
ORA #$10
STA $D011
```

5.5.2 Jel felírása szalagra

Valójában miből is áll ez a védelem? Abból, hogy a programunk mögé kiírunk egy jelsorozatot, és távolságukat később ellenőrizzük. Nyugodtan mondhatjuk, hogy a programunk másolását megakadályoztuk. Nézzünk meg egy példaprogramot:

```
C000 SEI
C001 LDA $D011
C004 AND #$EF
C006 STA $D011
C009 LDA #$10
C00B BIT $01
C00D BNE $C00B
C00F LDA $01
C011 AND #$DF
C013 STA $01
C015 LDA #$04
C017 LDX #$00
C019 DEY
C01A BNE $C019
C01C DEX
C01D BNE $C019
C01F SEC
C020 SBC #$01
C022 BNE $C019
C024 STX $FB
C026 LDA #$20
C028 STA $FC
C02A LDA $01
C02C AND #$F7
C02E STA $01
C030 LDX #$14
C032 DEX
C033 BNE $C032
C035 ORA #$08
C037 STA $01
C039 LDX #$3C
C03B DEX
C03C BNE $C03B
C03E DEC $FB
C040 BNE $C02A
C042 DEC $FC
C044 BNE $C02A
C046 LDA $D011
C049 ORA #$10
```

```
C04B STA $D011
C04E LDA $01
C050 ORA #$20
C052 STA $01
C054 STA $C0
C056 CLI
C057 RTS
```

Először tehát le kell tiltani az összes megszakítást, és ki kell kapcsolni a képernyőt. Ezután ellenőrizni kell, hogy minden rendben van-e, a gépkezelő lenyomta-e a megfelelő gombokat. A program azonban még ezután sem mehet tovább; várni kell egy kicsit, amíg a magnetofon motorja beáll a végleges sebességre. Ezért be kell építeni a programba egy késleltető ciklust, amely jelen esetben így néz ki:

```
    LDX #$00
11  DEY
    BNE 11
    DEX
    BNE 11
```

Amíg ez a ciklus lefut, bőven van ideje a motornak beállni.

A továbbiakban a $2000-es tárcímre beírtuk a felírandó jelek számát. Ez a szám első ránézésre nagynak tűnik. Egy jel ui. 400 ütemciklus, ami 2–3 s. Hogy miért 400 ütemciklus? Azért, mert 100 ütemig alacsony, utána 300 ütemig magas jelet küldünk. Ez összesen 400. A jelet 35–40 ütemciklusig kell tartani a motor ingadozása miatt. Különben félő, hogy hiba csúszik a műveletbe. A két jel között késleltetésre van szükség. Ezt a következő módon érhetjük el:

```
    LDY #$xx
11  DEY
    BNE 11
```

A gép xx-től fog számlálni. Ez az idő nekünk elég. A jelek felírása után befejezésképp kikapcsoljuk a motort, és engedélyezzük a megszakításokat. Bekapcsoljuk a képernyőt is. Figyelni kell arra is, hogy a $C0 értékre nullától különböző szám kerüljön, különben a magnetofon motorja a CLI parancs és a regiszterekbe való írás ellenére sem áll le.

Végül – egy másik programmal – vissza kell olvasni az adatokat.

```
C000 SEI
C001 LDA $D011
C004 AND #$FF
C006 STA $D011
C009 LDA #$FF
C00B STA $DC06
C00E STA $DC07
C011 LDA #$10
C013 BIT $01
C015 BNE $C013
```

```
C017 LDA $01
C019 AND #$DF
C01B STA $01
C01D LDX #$00
C01F DEY
C020 BNE $C01F
C022 DEX
C023 BNE $C01F
C025 LDA #$10
C027 BIT $DC0D
C02A BEQ $C027
C02C DEX
C02D BNE $C027
C02F STX $FB
C031 LDA #$18
C033 STA $FC
C035 LDA #$19
C037 STA $DC0F
C03A LDA #$10
C03C BIT $DC0D
C03F BEQ $C03C
C041 LDA #$08
C043 STA $DC0F
C046 LDA $DC06
C049 LDX $DC07
C04C LDY #$19
C04E STY $DC0F
C051 EOR #$FF
C053 TAY
C054 TXA
C055 EOR #$FF
C057 TAX
C058 CPY #$C8
C05A SBC #$00
C05C BCC $C065
C05E CPY #$58
C060 TXA
C061 SBC #$02
C063 BCC $C068
C065 JMP $FCE2
C068 DEC $FB
C06A BNE $C03A
C06C DEC $FC
C06A BNE $C03A
C070 LDA #$08
C072 STA $DC0F
C075 LDA $D011
C078 ORA #$10
C07A STA $D011
C07D STA $C0
C07F LDA $01
C081 ORA #$20
C083 STA $01
C058 CLI
C086 RTS
```

Az első pár sor ugyanaz. Letiltjuk a megszakításokat, ellenőrizzük a gépkezelőt, és megvárjuk, hogy a magnetofon felgyorsuljon. A timer értékét állítsuk maximumra ($FFFF). Mérésként mindig a pillanatnyi értéket kell leolvasni. Először várjunk ki kb. 256 jelet, nehogy rosszat ellenőrizzünk.

A leolvasást úgy tudjuk megvalósítani, hogy lekérdezzük a vezérlőregiszter ($DC0D) 4. bitjét. Ha ez 1, akkor ott a szalagon negatív él található. Mivel leolvasás után a regiszter törlődik, nincs vele egyelőre több dolgunk.

A jelek számlálóját a $FB/$FC címeken állítsuk $1800-ra. Ez ugyan kisebb a felírt értéknél, de nem szabad megfeledkezni arról, hogy a jelek kezdetét átugrottuk. Ezután kell a timert indítani:

```
LDA #$19; %00010011
STA $DC0F
```

A beállítás a következők szerint történik:

0. bit: a timer indul
3. bit: a timer lefutott
4. bit: a kezdeti érték kerüljön a számlálóba

Az 5. és 6. bitnek nullának kell lennie, mert belső ütemet kell számolni.

A timer lefutása után meg kell nézni a szalagjelet. Ha ez a keresett, akkor megállunk. Ha nem, újra kell indítani a timert. A kapott értéket invertáljuk, hogy az eltelt időt ütemciklusokban mérve kapjuk meg. Írásnál ez az érék 400 volt, így olvasás közben is megközelítőleg hasonló értéket kell kapnunk. Példánkban a jelek távolsága ± 200 ütemes szórást mutathat. Ha az észlelt távolság kisebb 200-nál, vagy nagyobb 600-nál, a védelem idegen behatolást jelez, a felírást idegennek tekinti, a vezérlést a RESET rutinra adja, amely a gépet alapállapotba hozza. Ha a leírtaknál kisebb az eltérés, tehát ha minden jel valódinak tekinthető, a program ugyanazt teszi, mint a felíró rutin: a motort kikapcsolja, a képernyőt bekapcsolja, és a megszakításokat engedélyezi. A lekérdezést tehát csak akkor zárjuk ezekkel a műveletekkel, ha a másolásvédelem rendben találkozott. Egyébként a program futása vagy RESET-tel zárul, vagy – ha a szalagon nincs semmi – végtelen várakozó ciklusba torkollik. (Ha az olvasó pontosan ugyanezt a másolásvédelmet akarja használni, nem szabad megfeledkeznie arról, hogy a tárolt programmal bárki szabadon rendelkezhet, aki e könyvnek birtokába jutott. A programot így csak módosításokkal érdemes felhasználni. Az egyik lehetőség pl. a jelek számának és távolságának megváltoztatása. Hogy milyen fokú legyen a módosítás mértéke, azt mindenki maga döntse el, programjainak fontosságától függően.)

Most egy kis segítséget nyújtunk a program kezeléséhez. Töltsük be a jeleket, az azonosító rutint a védeni szánt program alá. Ha magas szintű programozási nyelvvel dolgozunk (pl. BASIC), akkor egyszerűen írjuk a program mögé a DATA sorokat!

A felismerő rutint beépíthetjük bármilyen gépi kódú programba, tárbeli áthelyezése semmilyen gondot sem okoz. Lényegtelen, hogy milyen nyelvvel programozunk (BASIC, FORTH, gépi kód), érdemes a programot kódolni, autostarttal ellátni, és a RESET-et is célszerű letiltani és a felismerő rutinok végrehajtása után törölni, nehogy avatatlanok kezébe kerüljön programunk. Miután a programot felvettük szalagra, töltsük be a felíró rutint. Ha a DATA soros megoldást választottuk, akkor gépeljünk be egy RUN parancsot. Tegyük vissza a magnetofonba azt a szalagot, amelyen a védeni kívánt programunk van, indítsuk el a felíró rutint egy SYS 49152 paranccsal, és nyomjuk le a PLAY, RECORD gombokat. Három-négy perc múlva a képernyő visszanyeri eredeti színét, ami azt jelenti, hogy a programunk már védve van.

Most csévéljük vissza a szalagot a program elejére, és töltsük be a programot. Ha egy autostart is került a programba, akkor ezután minden betöltéskor automatikusan fog mozogni minden. Ha nem, indítsuk el a programot, de a PLAY gombot hagyjuk lenyomva. Az olvasórutin ui. most ellenőrzi a védelmet. A két program BASIC betöltője:

```
10 MU$="0123456789ABCDEF"
20 FORI=0TO89:READA$:B$=MID$(A$,2,1)
30 A$=LEFT$(A$,1)
40 FORH=1TO16
50 IFA$=MID$(MU$,H,1)THENA=H-1
60 IFB$=MID$(MU$,H,1)THENB=H-1
70 NEXT
80 A=A*16+B:POKE49152+I,A:NEXT:END
90 DATA 78          :REM   SEI
100 DATA AD,11,D0 :REM   LDA $D011
110 DATA 29,EF     :REM   AND #$EF
120 DATA 8D,11,D0 :REM   STA $D011
130 DATA A9,10     :REM   LDA #$10
140 DATA 24,01     :REM   BIT $01
150 DATA D0,FC     :REM   BNE $C00B
160 DATA A5,01     :REM   LDA $01
170 DATA 29,DF     :REM   AND #$DF
180 DATA 85,01     :REM   STA $01
190 DATA A9,04     :REM   LDA #$04
200 DATA A2,00     :REM   LDX #$00
210 DATA 88        :REM   DEY
220 DATA D0,FD     :REM   BNE $C019
230 DATA CA        :REM   DEX
240 DATA D0,FA     :REM   BNE $C019
250 DATA 38        :REM   SEC
260 DATA E9,01     :REM   SBC #$01
270 DATA D0,F5     :REM   BNE $C019
280 DATA 86,FB     :REM   STX $FB
290 DATA A9,20     :REM   LDA #$20
300 DATA 85,FC     :REM   STA $FC
310 DATA A5,01     :REM   LDA $01
320 DATA 29,F7     :REM   AND #$F7
330 DATA 85,01     :REM   STA $01
340 DATA A2,14     :REM   LDX #$14
350 DATA CA        :REM   DEX
360 DATA D0,FD     :REM   BNE $C032
```

```
370 DATA 09,08      :REM    ORA #$08
380 DATA 85,01      :REM    STA $01
390 DATA A2,3C      :REM    LDX #$3C
400 DATA CA         :REM    DEX
410 DATA D0,FD      :REM    BNE $C03B
420 DATA C6,FB      :REM    DEC $FB
430 DATA D0,E8      :REM    BNE $C02A
440 DATA C6,FC      :REM    DEC $FC
450 DATA D0,E4      :REM    BNE $C02A
460 DATA AD,11,D0   :REM    LDA $D011
470 DATA 09,10      :REM    ORA #$10
480 DATA 8D,11,D0   :REM    STA $D011
490 DATA A5,01      :REM    LDA $01
500 DATA 09,20      :REM    ORA #$20
510 DATA 85,01      :REM    STA $01
520 DATA 85,C0      :REM    STA $C0
530 DATA 58         :REM    CLI
540 DATA 60         :REM    RTS
550 DATA A4,A4      :REM    LDY $A4

READY.
```

a) Olvasórutin: $C000–$C086 (dec. 49152–49286)

```
99 MU$="0123456789ABCDEF"
100 FORI=0TO134:READA$:B$=MID$(A$,2,1)
101 A$=LEFT$(A$,1)
102 FORH=1TO16
103 IFA$=MID$(MU$,H,1)THENA=H-1
104 IFB$=MID$(MU$,H,1)THENB=H-1
105 NEXT
120 A=A*16+B:POKE49152+I,A:A=0:B=0:NEXT:END
300 DATA 78         :REM    SEI
301 DATA AD,11,D0   :REM    LDA $D011
302 DATA 29,EF      :REM    AND #$EF
303 DATA 8D,11,D0   :REM    STA $D011
304 DATA A9,FF      :REM    LDA #$FF
305 DATA 8D,06,DC   :REM    STA $DC06
306 DATA 8D,07,DC   :REM    STA $DC07
307 DATA A9,10      :REM    LDA #$10
308 DATA 24,01      :REM    BIT $01
309 DATA D0,FC      :REM    BNE $C013
310 DATA A5,01      :REM    LDA $01
311 DATA 29,DF      :REM    AND #$DF
312 DATA 85,01      :REM    STA $01
313 DATA A2,00      :REM    LDX #$00
314 DATA 88         :REM    DEY
315 DATA D0,FD      :REM    BNE $C01F
316 DATA CA         :REM    DEX
317 DATA D0,FA      :REM    BNE $C01F
318 DATA A9,10      :REM    LDA #$10
319 DATA 2C,0D,DC   :REM    BIT $DC0D
320 DATA F0,FB      :REM    BEQ $C027
321 DATA CA         :REM    DEX
322 DATA D0,F8      :REM    BNE $C027
323 DATA 86,FB      :REM    STX $FB
324 DATA A9,18      :REM    LDA #$18
325 DATA 85,FC      :REM    STA $FC
326 DATA A9,19      :REM    LDA #$19
327 DATA 8D,0F,DC   :REM    STA $DC0F
```

```
328 DATA A9,10    :REM  LDA #$10
329 DATA 2C,0D,DC:REM  BIT $DC0D
330 DATA F0,FB    :REM  BEQ $C03C
331 DATA A9,08    :REM  LDA #$08
332 DATA 8D,0F,DC:REM  STA $DC0F
333 DATA AD,06,DC:REM  LDA $DC06
334 DATA AE,07,DC:REM  LDX $DC07
335 DATA A0,19    :REM  LDY #$19
336 DATA 8C,0F,DC:REM  STY $DC0F
337 DATA 49,FF     :REM  EOR #$FF
338 DATA A8        :REM  TAY
339 DATA 8A        :REM  TXA
340 DATA 49,FF     :REM  EOR #$FF
341 DATA AA        :REM  TAX
342 DATA 70,C8     :REM  BVS $C022
343 DATA E9,00     :REM  SBC #$00
344 DATA 90,07     :REM  BCC $C065
345 DATA 70,58     :REM  BVS $C0B8
346 DATA 8A        :REM  TXA
347 DATA E9,02     :REM  SBC #$02
348 DATA 90,03     :REM  BCC $C068
349 DATA 4C,E2,FC :REM  JMP $FCE2
350 DATA C6,FB     :REM  DEC $FB
351 DATA D0,CE     :REM  BNE $C03A
352 DATA C6,FC     :REM  DEC $FC
353 DATA D0,CA     :REM  BNE $C03A
354 DATA A9,08     :REM  LDA #$08
355 DATA 8D,0F,DC :REM  STA $DC0F
356 DATA AD,11,D0 :REM  LDA $D011
357 DATA 09,10     :REM  ORA #$10
358 DATA 8D,11,D0 :REM  STA $D011
359 DATA 85,C0     :REM  STA $C0
360 DATA A5,01     :REM  LDA $01
361 DATA 09,20     :REM  ORA #$20
362 DATA 85,01     :REM  STA $01
363 DATA 58        :REM  CLI
364 DATA 60        :REM  RTS

READY.
```

b) Felírórutin: $C000–$C057 (dec. 49152–49239)

Védelmi rendszerünkkel nagymértékben meg lehetünk elégedve, de nem szabad megfeledkezünk hátrányairól sem, pl. a betöltési sebesség jelentősen csökkent. (Gondolkozzunk el a felmerülő problémán: hogyan lehetne a jeleket gyorsabban szalagra küldeni?) Fejezetünk elején tettük azt a kiábrándító megállapítást, hogy képtelenség általános másolóprogramot készíteni.

Elérkezett az idő, hogy megállapításunkat megerősítsük. A szalagra felírt adatok egymástól meghatározott távolságra (időtartamra) lévő jelekből állnak. Másolóprogramunk annyira nem „intelligens", hogy tudja, egy vagy több jeltávolsággal dolgozunk, elvileg csak annyit tehet, hogy mér és tárolja az egyik jeltől a másik jelig eltelt időtartamot. Ez a megoldás viszont „felfalná" memóriánkat, jóval nagyobb volna a helyigénye, mint amennyi egy átlagos program betöltése után a tárban szabadon marad. Egy szemléletes példával rögtön átlátjuk e megoldás hátulütőit.

Tegyük fel, hogy a felírás formátuma két különböző időtartamra épül úgy, hogy töröljük az egyik nullát, a másik az egyet jelöli. Minden egymást követő nyolc bit után egy további (az előző kettőtől különböző hosszúságú) jel áll. Ahhoz, hogy gépünk ezt le tudja mérni, jelenként két byte-ra lenne szükség. Ha a másolandó program – tegyük fel – 4 Kbyte-os, a szalagon összesen 4096*(8+1) = 36864 byte található. A másolóprogramnak tehát összesen 2*36864 byte-ra lenne szüksége, ami már eleve jóval több a C 64-es memóriájánál. Ráadásul nem tekinthető megoldásnak az sem, hogy a programot részekre bontjuk, hiszen az összefüggő adatsorozatot összefüggően kell tárolni; vagy beletörődünk abba, hogy a másolóprogram csak igen rövid programokra működik, vagy legalább 256 Kbyte-os tárbővítést kell használnunk. Olyan másolót egészen biztosan tudnánk készíteni, ami az átjátszó zsinóros megoldást szimulálja. De a beolvasott jeltávolságok a technikai pontatlanságok miatt sohasem egyeznek meg tökéletesen a felírt jeltávolságokkal, úgy is fogalmazhatnánk, hogy aszinkronban lesznek egymással a jelek. Ez pedig azt eredményezi, hogy a másolatok minősége fokozatosan romlik, míg egyszer teljesen hasznavehetetlen nem lesz.

Végül is visszajutunk a kiindulóponthoz: a másolóprogram csak akkor lehet tökéletes, ha ismeri a felírás formátumát; az pedig teljességgel lehetetlen, hogy valaki egy programon belül az összes lehetséges formátumot figyelembe vegye.

5.5.3 Új, kazettás felírási formátum: KMR

Az előző fejezetben felmerült az az ötlet, hogy a megszokott kazettára való írás helyett egy egyedi, új rögzítési formát dolgozzunk ki mágnesszalagra. Ezt az ötletet kár lenne elvetnünk, lássunk hozzá a megvalósításhoz! Nevezzük a programot KMR-nek (kazettás másolásvédelmi rendszer). Új rendszerünk hatásossága érdekében a következő követelményeknek kell megfelelnie:

1. Létező másolóprogramokkal ne legyen másolható.
2. Nagy felírási sebességgel működjön.
3. A betöltő gyors legyen és automatikusan induljon, és a betöltött programot is automatikusan indítsa.
4. Kezelése egyszerű legyen azok számára is, akik részletesen nem kívánnak foglalkozni a rendszer működésével.
5. Legyen elhelyezhető a RAM-ban a BASIC interpreter alatt.
6. Legyen opcionálisan áthelyezhető a RAM-ban.

A program működési alapelve a következő: az operációs rendszer SAVE rutinját a vektorok segítségével saját rutinnal helyettesítjük.
A SAVE"programnév" parancs hatására először a betöltőprogram egy részét eltoljuk a $02A8 (680) címre.

A program többi részét a file-név helyén tároljuk, így azok a kazettapuffer betöltésekor kerülnek a tárba. A betöltőrutint a már megszokott módon, „lassan" tároljuk úgy, hogy autostart is legyen a rutinban.

E rutin mögé kerül a tulajdonképpeni program, új formátumban. Betöltéskor a gépkezelőnek a megszokott módon be kell írnia egy LOAD parancsot, és le kell nyomnia a kazettás egyégen a PLAY feliratú gombot. Ekkor az operációs rendszer behívja az új betöltőt, és el is indítja. A betöltő ugyanezt teszi a főprogrammal.

Mi is az új formátum lényege? A szalag elejére felírunk egy szinkronjelet, ami a program kezdetét mutatja. A szinkronjel mögé felírjuk a program kezdő- és végcímét, majd ezt követően magát a programot. A sort az a byte zárja, amelyben az ellenőrző összeget tároljuk. Az ellenőrző összeg tájékoztat a betöltési hibákról. Kiszámítása rendkívül egyszerű, nem kíván komolyabb ismereteket. A program utasításai között rendre EOR műveletet végzünk. Miután beolvastuk az adatokat, a kapott értéket össze kell hasonlítanunk az eredetivel, és ha a két érték között különbség van, betöltési hiba történt. A leírtak feltételezik, hogy a bitek és a byte-ok szalagra írásának módszerét már kidolgoztuk.

A tárolás alapelve, hogy a magas bit (1-es) távolabb van az előző jeltől, mint az alacsony bit (0-ás). Az 1 értékű bitnek 643, a 0 értékű bitnek pedig 388 ütemciklusnyi távolság felel meg. Egy byte természetesen 8 egymást követő bitből áll.

A betöltőprogram készítése közben a felmerülő probléma a következő: hogyan ismerjük fel a főprogram legelső bitjét? A válasz kézenfekvő: a kulcsszó a szinkronjel. A szinkronjel azonos, előre rögzített tartalmú byte-ok sorozata. Nem volna ésszerű megoldás pl. a $AA (%10101010) értéket választanunk, hiszen ha több ilyen jelet írtunk egymás mellé, a bitkombinációnk összefolyik.

...1010101010101010...

Ebben az esetben szinte képtelenség lenne elkülöníteni, hogy valójában melyik a szinkronjel első bitje, hiszen az ugyanúgy lehet a harmadik, mint az ötödik.

Sokkal nyilvánvalóbb és áttekinthetőbb az a megoldás, amelyet most ábrázolunk:

$C0 (%11000000)
...0011000000110000000110000...

Ezt a sorozatot nagyon egyszerűen byte-okká törhetjük:

...00 11000000 11000000 110000...

Szokás még a szinkronjel kiegészítése egy vagy több jól felismerhető byte-tal. Ezek használata biztonságosabbá teszi a programkezdet felismerését, nehogy összetévesszük valamilyen más szalagjellel. Példánkban erre a $48 értékű byte-ot használjuk.

Hogyan tudunk pontosan időzíteni írás és olvasás közben? A válasz: egy megszakítási rutinnal. Írás közben a megszakítást a CIA (Control Interface Adapter) timer váltja ki. A timer értéke állandó. Mielőtt kiváltanánk a megszakítást, meg kell vizsgálnunk, hogy a felírandó bit nulla-e vagy sem. Ha a bit nulla, akkor azt azonnal felírjuk a szalagra; ha egy, akkor egy várakozóciklust indítunk el, mielőtt felírnánk a jelet. Ekkor a timert újra elindítjuk. A timer, míg olvasunk, hasonló szerepet kap. Most azonban nem megszakítást (BREAK) hoz létre, hanem a szalagon észlelt jel. Az a bit, amely a timer lefutását jelzi, azonos a beolvasott bittel, hiszen értéke csak akkor lesz 1, ha a két jel közötti távolság nagyobb, mint egy rögzített érték.

Az írás és olvasás közben eltelt időt a KMR a következőképpen nézi: ha a $0314/$0315 címeken található IRQ vektorra hivatkozunk, a megszakítás kiváltása és a saját megszakítórutin elérése közben átlagosan 39 ütemciklus telik el.

Ahhoz, hogy felírjunk egy 0 bitet, a felírórutinnak 16 további ciklusra van szüksége, hogy a jelet vezetékre küldje, és még másik 6 ciklusra, hogy a timert újraindítsa. Ha a timert a 327 ($0147) értékről indítjuk, a nulla bithez tartozó időtartam:

A timer elindítása:	6 ciklus
Az IRQ-ig eltelt idő:	327 ciklus
A saját IRQ rutin eléréséhez:	39 ciklus
A jel továbbításához:	16 ciklus
Összesen:	388 ciklus

Ahhoz, hogy egy 1-es bitet felírhassunk, ezt az értéket meg kell növelnünk 255 ütemciklussal, mert így az előző jeltől mért távolság 388 + 255 = 643 ütemciklus lesz. Attól függetlenül, hogy a bit alacsony vagy magas, az íróvezeték kb. 80 ütemciklus erejéig marad alacsony jelszinten, ami az időmegszakítás szempontjából nézve, teljesen lényegtelen esemény.

Olvasásnál a timert úgy kell beállítani, hogy rövid szalagjel esetén a vezérlést megszakító regiszter leolvasása alatt ne fusson le, hosszú jel esetén pedig elég időt hagyjon arra, hogy az ICR-ben a timerbitet 1-re állítsuk.

Az alacsony állapotú bit leolvasásához szükséges időt a következő módon határozhatjuk meg.

Kezdőpont: a negatív él megszakítást vált ki.

A megszakítási rutin eléréséhez:	39 ciklus
Az ICR leolvasásához:	4 ciklus
A timer indításához:	6 ciklus
Összesen:	49 ciklus

A következő jel eléréséig:	388-49 = 339 ciklus
Az ICR leolvasásáig:	39+ 4 = 43 ciklus
Összesen:	382 ciklus

Ha ugyanezt a számítást elvégezzük az 1-es bitre, a végeredmény 637 ciklus.

A timer startértékét úgy kell megállapítani, hogy a kapott két érték közé essen, azaz pl. 512 ($0200) legyen.

A következő kérdés, hogyan kerülnek a byte-ok a szalagra? A tárolás megszakítórutinja mindig csak 1 byte-ot ír fel. E művelet közben 4 regiszterre lesz szükségünk:

- egy eltolóregiszterre, amelyből a következő bitet átvesszük;
- egy olyan számlálóra, amely az átvitt bitek számát mutatja meg;
- egy átmeneti tárolóra, amelyben elhelyezzük a következő byte-ot;
- egy kapcsolóra, amely megmutatja, hogy a megszakítási rutin beolvasta-e már az átmeneti tároló tartalmát.

A megszakítási rutinnal párhuzamos futó főprogram addig vár, amíg ez a rutin az átmeneti tárolóból a byte-ot be nem olvasta, és a soron következő byte-ot csak ekkor írja bele. A kapcsolót használat előtt a rutin $80 (128)-ra állítja, és a kapcsoló későbbi értéke alapján dönti el a főprogram, hogy az átmeneti tároló szabad-e már. A kapcsoló 0 értéke azt jelzi, hogy az átmeneti tár ismét foglalt. Ahogy a byte összes bitje felíródott a szalagra, a megszakítási rutin az átmeneti tároló tartalmát átviszi az eltolási regiszterbe, és ezzel a kör bezárult.

Ugyanezen regiszterek segítségével történik az olvasás is, csak minden pontosan fordított sorrendben. A beolvasott bitet először az eltolási regiszterbe viszi. Miután mind a 8 bitet beolvasta, az eltolási regiszter tartalmát átviszi az átmeneti tárolóba. A beolvasott byte átadása, a főprogram és a rutin közötti kommunikáció ugyanígy zajlik le, mint az írásnál.

A szinkronizáció, azaz a legelső, már az adathoz tartozó jel azonosítása nagyon fontos dolog, amire nagy hangsúlyt kell fektetnünk. A főprogram egy ideig hagyja dolgozni a rutint anélkül, hogy az adatot az átmeneti tárolóból átvenné. Lekérdezi az eltolási regisztert, és megvizsgálja, hogy értéke $C0 (192 – szinkronjel) vagy sem. Ha igen, akkor 7-re, azaz maximumra állítja a beolvasott bitek számát jelző változatot, még mielőtt a következő megszakítási kérelem megérkezne (a számláló mindig héttől nulláig folyamatosan csökken). Az adatbyte beolvasására csak ezután kerülhet sor.

A fejezet elején hat alapvető követelményt támasztottunk a KMR-rel szemben. Az első követelmény, hogy a program gyakorlatilag másolhatatlan legyen. A lényegesen új tárolási formátum létrehozásával ezt már teljesítettük.

A második pont a felírási sebességre vonatkozott. A sebességet a jelek közötti távolság határozza meg. Esetünkben ez 516 ütemciklus. A felírási sebesség 0,98 MHz ütemfrekvencia mellett kb. 1900 bitet jelent másodpercenként (azaz 1900 baud). Ez a sebesség kb. hatszor akkora, mint amekkorát az operációs rendszer által használt formátum segítségével elérhetünk. Ez a hatszoros sebesség nem a felső határ. Ha valakinek ez mégsem felelne meg, akkor az assembler programot javítsa ki a következő három helyen:

$C0C9 – a timer kezdőértéke írásnál,
$C148 – a nulla és egy közötti különbség időciklusa,
$037D – a timer kezdőértéke olvasásánál.

A nagy sebesség viszont nagymértékben növelheti a hiba előfordulásának gyakoriságát. Ezt mindig szem előtt kell tartanunk. Lehet megoldás persze, hogy a C 64-eshez kazettás egység helyett hi-fi berendezést illesztünk. (A gyakorlati határ kb. 3600 baud, de folytak biztató kísérletek 7200 baudon is.)

A betöltőrutin automatikus indítását a $0302/$0303 (770/771) BASIC RUN rutin aktivizálásával indítjuk (ld. 3 fejezet):

```
LDA #$00
JSR $A871
JMP $A7AE
```

Ha programot nem RUN-nal, hanem SYS-szel akarjuk indítani, akkor helyettesítenünk kell a hívást közvetlen ugrással, a JMP utasítással. Új rendszerünk kezelését a SAVE vektor átírása rendkívül megkönnyíti. Előnye, hogy a program bármilyen monitorprogrammal is képes együttműködni.

Kell még néhány szót szólnunk a KMR kezeléséről. Írjuk be és indítsuk el a BASIC betöltőt. Ezután a SYS49152 (49152 = $C000) parancs kiadásával aktivizáljuk a KMR-t. Töltsük be a védeni szánt programot a tárba, és mentsük a szalagra a `SAVE"programnév"` paranccsal. Ha nem akarunk minden egyes használatkor vesződni a BASIC betöltővel, a gépi kódú programot a betöltő lefutása után szintén tárolnunk kell egy programmal:

```
109B  11 00 00 00 00 00 00 00
10A3  00 00 00 00 00 00 00 00
10AB  00 AC 7F 00 00 00 00 00
1100  74 A4 74 A4 C3 C2 CD 38
1108  30 00 00 00 00 00 00 00
```

Ezután a védelmi rendszer betöltése következik: `LOAD"KMR"`, és installáljuk az előbbi SYS paranccsal. Ilyenkor ajánlatos egy NEW parancs kiadása is, hogy a BASIC mutatókat alapállapotba hozzuk (ez természetesen nem törli a gépi kódú programot). Figyelem: a program ugyan átállítja a SAVE vektorokat, de a STOP/RESTORE ismét alaphelyzetbe hozza. Ha ezt a két billentyűt használtuk, akkor a programot még egyszer le kell futtatnunk egy SYS 49152 paranccsal.

Ahhoz, hogy a KMR képes legyen lementeni a BASIC interpreter alatt a programot a RAM területéről, az 1-es tárcímre $06-ot kell írnia. A tárcím a módosítás után azonnal ismét alaphelyzetbe kerül.

Új rendszerünk csak akkor alkalmazható 38 Kbyte-nál hosszabb programra, ha nem használunk file-nevet. Eljárásunk "Out of Memory Error" üzenettel végződik. Néhány ötlet, mint pl. a `POKE 56,208:CLR` parancs, segítségünkre

lehet. Hatására a BASIC interpreter a karakterláncokat egy szabad tárterületen helyezi el. A védelmi eljárás befejeztével az alapállapotot pl. egy BASIC program futtatásához a `POKE 56,160:CLR` paranccsal állíthatjuk vissza.

A hatodik pontban a program áthelyezhetőségét szabtuk meg követelményként. Ehhez elég a rendszerünk apró módosítása. Mivel a KMR tárolásánál a betöltési és a kezdőcímet külön kezeli, jogunkban áll helyükre valóban eltérő értékeket írni. A szükséges helyet két üres utasítással (NOP – No Operation) biztosíthatjuk. Ha a betöltési címet módosítani szeretnénk, a NOP utasítások helyére a megfelelő értékeket kell beírnunk.

`C07E LDA#$` a betöltési cím felső byte-ja

`C085 LDA#$` a betöltési cím alsó byte-ja

Ez a változtatás több szempontból is előnyös. Egyébként lehetővé teszi, hogy azokat a programrészeket is kimentsük, amelyeket olyan tárterületre kell betölteni, ahonnan nem lehet kimenteni. Ilyenek pl. a következő területek: $C000-tól $C269, itt található új kazettás rendszerünk, és a $E000–$FFFF az operációs rendszer által foglalt területek. A kazettapuffer sokrétűségét mi sem bizonyítja jobban, hogy saját betöltőrutinunkat is ide helyeztük el. A $0400 cím fölött már nincs helyigényünk, ezzel a szabad területtel szabadon rendelkezhetünk. Ha egy programot pl. a $C000–$CFFF-ig terjedő területen tároltunk, először helyezzük át egy monitorprogram segítségével egy szabad területre, pl. $4000–$4FFF-re.

Most töltsük be rendszerünket, és módosítsuk a következő sorokat:

`C07E LDA #$C0` – $C000 felső byte-ja

`C085 LDA #$00` – $C000 alsó byte-ja

Ezután indítsuk el a KMR-t a SYS 49152 paranccsal. Most már nyugodtan elmentheti a programot a $4000–$4FFF területről. A betöltéskor olyan lesz, mintha a tárolást $C000-tól kezdtük volna.

Említettük, hogy a módosítás más szempontból is előnyös. Lehetőségünk van egy olyan programvédelem kialakításra, amely egyébként csak lemezen tárolt programra alkalmazható. A $AC/$AD címeket betöltés közben mutatóként használjuk, a következő byte kijelölésére (ld. az assembler listát). Ha ezt a mutatót megváltoztatjuk, a következő byte-ok tökéletesen máshová kerülnek. Feltétlenül szükséges vigyáznunk, amikor a $9B címet elérjük, ez a cím ui. tartalmazza az ellenőrző összeget. Ha ez az összeg a betöltés végén nem megfelelő, rendszerünk feltételezi, hogy az adatok rosszak, és a RESET-re ugrik.

Tegyük fel, hogy a tárban a 6. ábrán látható byte-sorozat található.

```
109B  11 00 00 00 00 00 00 00
10A3  00 00 00 00 00 00 00 00
10AB  00 AC 7F 00 00 00 00 00
1100  74 A4 74 A4 C3 C2 CD 38
1108  30 00 00 00 00 00 00 00
```

Tároljuk a $109B–$1109 úgy, hogy tartalma a $009B-re töltődjön vissza. Ellenőrző összegünk jelen esetben hibás lesz, de nem kell megijedni, szándékosan tettük ezt. A következő nulla byte-ok hatástalanok. Az első fontos byte a $AC, amely a $00AC címre kerül (tehát a nulláslapra). A rendszer, miután beírta a $AC-t a $00AC címre, a mutató értékét megnöveli. A következő byte tartalma $7F, ez a $00AD címre kerül, és a mutató jelen esetben a $7FAE címen áll. A soron következő byte-ok betöltése itt kezdődik. A $8000-es címen a "CBM80" azonosító található (ld. programvédelemről szóló fejezet). A betöltés befejezése után a KMR megvizsgálja, hogy az ellenőrző összeg helyes-e. Ha nem, akkor a RESET-re ugrik, ellenkező esetben a processzor a CBM80 szöveg azonosítása után, a BASIC értelmezőre (a $A474 címre) adja a vezérlést. Az ugrási címet kell csak módosítanunk, ha ehelyett a főprogramot akarjuk elindítani.

A fejezet hátralévő részében a védelmi rendszer assembler listáit adjuk közre, mellékelve a fontosabb tárcímek és rutinok dokumentációit. E fejezetben leírtak feltehetőleg meggyőzték az olvasót, és felkeltettek további érdeklődést a „lemásolhatatlan program" megvalósítása iránt.

A kazettás másolásvédelmi rendszer (KMR) assembler listája

A hivatkozott tárcímek:

$98: a betöltött/tárolt program ellenőrző összege
$9C: ha negatív az értéke, a byte-ot beolvassuk/kiírjuk
$AC/$AD: mutató az aktuális byte-ra (a $AC az alsó, a $AD a felső byte)
$AE/$AF: mutató a végcím + 1-re
$B4: a beolvasott/kiírt bitek számlálója
$BE: az aktuális byte
$BF: eltolási regiszter a byte beolvasásához/kiíráshoz
$C1/$C2: a kiírt betöltési cím

A SAVE rutin:

```
C000  LDA #$08      A SAVE rutin megváltoztatása
C002  STA $0332     A $033C-be.A SAVE vektor alsó byte-ja
C005  LDA #$C0      A $C008 felső byte-ja.
C007  STA $0333     A $0333-ba a SAVE vektor magas byte-ja
C00A  RTS           Visszatérés
C00B  LDA $BA       Eszközszám
C00D  CMP #$01      Ha 1, akkor kazetta
C00F  BEQ $C014     Ha igen, akkor ugrás a $C014-es címre
C011  JMP $F5ED     Ha nem, ugrás a normál SAVE rutinra
C014  LDX #$00      Az eltolás elökészítése
C016  LDA $C23E,X   A betöltörutin része
C019  STA $02AB,X   Eltolása $02AB-tól kezdödöen
C01C  INX           Számláló növelése
C01D  CPX #$48      Be van olvasva az összes byte?
C01F  BNE $C016     Elágazás, ha nem ugrás vissza $C016-ra
```

```
C021  LDY #$00     Mutató a filenév előkészítésre
C023  CPY $87
C025  BEQ $C031    Elágazás, ha igen
C027  LDA ($BB),Y  A filenév byte-ok elvitele
C029  STA $C183,Y  Elhelyezés a töltőprogram elé
C02C  INY          Számláló növelése
C02D  CPY #$10     16-tal való összehasonlítás
C02F  BNE $C023    Ha nem, ugrás vissza a $C023-as címre
C031  CPY #$10     16-tal való összehasonlítás
C033  BEQ $C03E    Ha igen, ugrás a $C03E címre
C035  LDA #$20     Ha nem feltöltés szóközökkel
C037  STA $C183,Y  Elhelyezés a file-név mögé
C03A  INY          A számláló növelése
C03B  JMP $C031    Ugrás a $C031-es címre
C03E  LDA #$01     File-szám
C040  TAX          Eszközszám legyen 1
C041  TAY          Másodlagos cím (=1,nincs elmozdítás)
C042  JSR $FFBA    Ugrás a FILPAR rutinra
C045  LDA #$BC     File-név hossza
C047  LDX #$83     File-név folytatása
C049  LDY #$C1     Felső byte-ja a $C183-as címnek
C04B  JSR $FFBD    FILNAM
C04E  LDA $AE      A végcím alsó byte-ja
C050  PHA          A veremből az akkumulátorba
C051  LDA $AF      A végcím felső byte-ja
C053  PHA
C054  LDA $C1      A kezdőcím alsó byte-ja
C056  PHA
C057  LDA $C2      A kezdőcím felső byte-ja
C059  PHA
C05A  LDA #$A8     $02A8 az új
C05C  STA $C1      kezdőcím
C05E  LDA #$02     a $C1/$C2 byte-okba
C060  STA $C2
C062  LDA #$04     $0304 az új
C064  STA $AE      végcím
C066  LDA #$03     a $AE/$AF byte-okba
C068  STA $AF      Indulási címet
C06A  LDA #$D3     az autostarthoz
C06C  STA $0302    a $0302/$0303 címekre viszi
C06F  LDA #$02     BASIC vektor, egy sor bevitelére
C071  STA $0303    0-át tesz a $9D-be
C074  LDA #$00
C076  STA $9D
C078  JSR $F5ED    A normál SAVE rutin hívása
C07B  PLA          A kezdőcím magas byte-j nak
C07C  STA $AD      visszahozása
C07E  NOP          No OPeration
C07F  NOP
C080  STA $C2      A töltési cim magas byte-ja
C082  PLA          Vissza a verembe
C083  STA $AC
C085  NOP
C086  NOP
C087  STA $C1
C089  PLA          A végcím magas byte-ja
C08A  STA $AF
C08C  PLA          A végcím alacsony byte-ja
C08D  STA $AE
C08F  JSR $E453    A BASIC vektorok visszaállítása
```

```
C092  SEI          A megszakítás letiltása
C093  LDA $D011    Megszakítás a VIC-től
C096  AND #$EF     A képernyő kikapcsolása
C098  STA $D011
C09B  LDA #$44     Megszakítási vektor
C09D  STA $0314    a $C144-re
C0A0  LDA #$C1     irányítva
C0A2  STA $0315
C0A5  LDA $01
C0A7  PHA          Tárolás a veremben
C0A8  LDA #$06     A saját konfiguráció elhelyezése
C0AA  STA $01      Emellett egyidejüleg a magnó
C0AC  STA $C0
C0AE  LDA #$03     szalagidő elkészítése
C0B0  LDY #$00
C0B2  STY $9B      Ellenörzőösszeg nullától
C0B4  DEX          X regiszter tartalmának csökkentése
C0B5  BNE $C0B4
C0B7  DEY
C0B8  BNE $C0B4
C0BA  SEC
C0BB  SBC #$01
C0BD  BNE $C0B4
C0BF  LDA #$7F     A CIA 1 összes megszakítási
C0C1  STA $DC0D    lehetőségének kikapcsolása
C0C4  LDA #$82     Megszakítás a Timer B-n keresztül
C0C6  STA $DC0D    Engedélyezve a
C0C9  LDA #$47     Timer B
C0CB  STA $DC06    értékétől
C0CE  INX
C0CF  STX $DC07
C0D2  LDA #$07     Bitszámláló
C0D4  STA $B4      hetes értékre állítja
C0D6  LDA #$19     a Timer indulásakor
C0D8  STA $DC0F
C0DB  LDA $DC0D
C0DE  CLI          Megszakítás engedélyezése
C0DF  LDA #$C0     A 192-es szinkronjel
C0E1  JSR $C17A    írása szalagra
C0E4  DEY
C0E5  BNE $C0DF    Ha nem
C0E7  LDA #$48     $48 a programkezdőjel
C0E9  JSR $C17A    Felírása kazettára
C0EC  LDA $C1      A kezdőcím alsó byte-ja
C0EE  JSR $C17A    Ismét felírás
C0F1  LDA $C2      A kezdőcím felső byte-ja
C0F3  JSR $C17A    Felírás
C0F6  LDA $AE      A végcím tárolása a tárolóba
C0F8  SEC          Levonunk $AE-t
C0F9  SBC $AC      a kezdőcím alsó byte-jából
C0FB  PHA          Tárolás
C0FC  LDA $AF      Levonunk $AF-et
C0FE  SBC $AD      a kezdőcím felső byte-jából
C100  TAX          Tárolás
C101  PLA          Programhossz alsó byte-ja
C102  CLC          Hozzáadjuk a kezdőcím
C103  ADC $C1      alsó byte-ját a töltőprgnak
C105  JSR $C17A    Felírás kazettára
C108  TXA          Programhossz felő byte-ja
C109  ADC $C2      Hozzáadjuk a kezdőcím felső byte-ját
```

```
C10B  JSR $C17A    Felírás kazettára
C10E  LDA ($AC),Y  Programbyte elhozatala (Y=0)
C110  TAX          Közbülső adatok tárolása
C111  EOR $9B      Ellenőrzőösszeg képzése
C113  STA $9B      és titkos tárolás
C115  TXA
C116  JSR $C17A    Kazettára írás
C119  JSR $FCDB    Folv. növelése a $AC/$AD mutatóknak
C11C  JSR $FCD1    egészen a végcím $AE/$AF eléréséig
C11F  BNE $C10E    Ha még nem értük el, ugrás vissza
C121  LDA $9B      Ellenörzőösszeg elhozása
C123  JSR $C17A    Irása kazettára
C126  JSR $C17A    Várunk az utolsó byte-ig
C129  JSR $C17A
C12C  SEI          Megszakítás letiltása
C12D  PLA
C12E  ORA #$20     Motor kikapcsolása
C130  STA $01      A régi vektorok visszaállítása
C132  JSR $FD15    Megszakításvektor visszaállítása
C135  JSR $FDA3    CIA-t visszaállítjuk a régi üzemre
C138  JSR $FC93    A motort kikapcsoljuk, a képernyőt be
C13B  LDA #$0E     Visszaállítjuk a képernyő
C13D  STA $D020    eredeti színét
C140  CLI          Megszakítás engedélyezve
C141  JMP $A474    Vezérlés átadása a BASIC értelmezőnek
```

Megszakítás rutin a szalagra íráshoz:

```
C144 ASL $BF      Egy bit az eltolóregiszterből
C146 BCC $C14D    Ugrás, ha a bit nulla
C148 LDX #$33
C14A DEX          Még 255 ütemciklus várás
C14B BNE $C14A    Ugrás, ameddig nincs meg a 255
C14D LDA $01      Kapubyte bevitele
C14F AND #$F7     3. bit kinullázása
C151 STA $01
C153 LDX #$19
C155 STX $DC0F
C158 LDX #$10
C15A DEX          80 ütemciklusig várás
C15B BNE $C15A    Ugrás, ha még nincs meg
C15D ORA #$08     3. bit egyesre állítása
C15F STA $01      Kontrollként a képernyő
C161 INC $D020    színének változtatása
C164 DEC $B4      A komplett byte már be van írva?
C166 BPL $C174    Ha nem, akkor ugrás
C168 LDA #$07     Bitszámláló beállítása
C16A STA $B4
C16C LDA $BE
C16E STA $DF
C170 LDA #$80
C172 STA $9C
C174 LDA $DC0D    Megszakítás szabaddá tétele
C177 JMP $FEBC    Megszakítás vége
```

Egy byte kiírása:

```
C17A BIT $9C     Utolsó byte már be van olvasva?
C17C BPL $C17A   Ha nem, ugrás vissza
C17E STA $BE     Legközelebbi byte átadása
C180 STY $9C     Signal byte küldése
C182 RTS         Visszatérés a szubrutinból
```

$C183-tól $C192-ig: a file-név számára fenntartott hely. A következő rutint ($C193-tól) betöltéskor áthelyezzük a $0351 címre.

```
0351 SEI
0352 LDA #$04
0354 STA $D011
0357 LDA #$A8
0359 STA $0314
035C LDA #$02
035E STA $0315
0361 LDA $01
0363 AND #$1F
0365 STA $01
0367 STA $C0
0369 LDY #$00
036B STY $9D
036D DEX
036E BNE $036D
0370 DEY
0371 BNE $036D
0373 LDA #$7F
0375 STA $DC0D
0378 LDA #$90
037A STA $DC0D
037D LDA #$FE
037F STA $DC06
0382 LDA #$01
0384 STA $DC07
0387 LDA #$19
0389 STA $DC0F
038C LDA $DC0D
038F CLI
0390 LDA $BF
0392 CMP #$03
0394 BNE $0390
0396 LDA #$07
0398 STA $B4
039A LDX #$10
039C STX $9C
039E JSR $02CA
03A1 CMP #$03
03A3 BNE $0390
03A5 DEX
03A6 BNE $039E
03A8 JSR $02CA
03AB CMP #$03
03AD BEQ $03A8
03AF CMP #$48
03B1 BNE $0390
03B3 JSR $02CA
```

```
03B6 STA $AC,X
03B8 INX
03B9 CPX #$04
03BB BNE $03B3
03BD JSR $02CA
03C0 STA ($AC),Y
03C2 EOR $9B
03C4 STA $9B
03C6 JSR $FCDB
03C9 JSR $FCD1
03CC BNE $03BD
03CE JSR $02CA
03D1 CMP $9B
03D3 BEQ $03D8
03D5 JMP $FCE2
03D8 SEI
03D9 JSR $FD15
03DC JSR $FDA3
03DF JSR $FC93
03E2 LDA #$0E
03E4 STA $D020
03E7 LDX $AE
03E9 LDY $AF
03EB CLI
03EC JMP $02D9
03EF LDA #$E1
03F1 STA $0328
03F4 LDA #$00
03F6 JSR $A871
03F9 JMP $A7AE
```

Ugyanezt tesszük a következő rutinnal is ($C23F-től), áthelyezzük a $02A8 címre.

Megszakítás rutin a betöltéshez:

```
02A8  LDA $DC0D   Megszakítás-regiszter olvasása
02AB  LDX #$19    A Timer újra indul
02AD  STX $DC0F
02B0  LSR         Egy bit a Timer B-nek
02B1  LSR         A Carry bitbe tolja
02B2  ROL $BF     A bit az eltolási regiszterbe jut
02B4  INC $D020   Kontroll
02B7  DEC $B4     Még nincs egy byte?
02B9  BPL $02C7   Ha nincs, ugrás
02BB  LDA #$07    A bitszámláló héttől
02BD  STA $B4     csökkenti az értéket
02BF  LDA $BF     nulláig
02C1  STA $BE
02C3  LDA #$80    Szinkronjel
02C5  STA $9C
02C7  JMP $FEBC   Megszakítás vége
```

Byte tárolása beolvasás után:

```
02CA BIT $9C      Van még byte?
02CC BPL $02CA    Ha nincs, akkor
02CE STY $9C      szinkronjel kitörlése
02D0 LDA $BE      Byte elvitele
02D2 RTS          Visszatérés a rutinból
02D3 JSR $E453    BASIC vektorok visszaállítása
02D6 JMP $0351    Ugrás a betöltő rutinra
02D9 STX $2D      BASIC program vége mutató (alcsony byte)
02DB STY $2E      magas byte
02DE JSR $A569    CLR utasítás
02E1 JSR $A533    ugrás a $A533 címen levő szubrutinra
02E4 JMP $03EF    ugrás $03EF-re
```

A KMR rendszer teljes programlistája:

```
100 FORI=1TO636STEP15:FORJ=0TO14
101 READA$:A$=LEFT$(A$,2):B$=RIGHT$(A$,1)
105 A=ASC(A$)-48:IFA>9THENA=A-7
110 B=ASC(B$)-48:IFB>9THENB=B-7
120 A=A*16+B:POKE49151+I+J,A:NEXTJ:NEXTI:END
300 DATA  A9,0B     :REM LDA #$0B
301 DATA  8D,32,03  :REM STA $0332
302 DATA  A9,C0     :REM LDA #$C0
303 DATA  8D,33,03  :REM STA $0333
305 DATA  A5,BA     :REM LDA $BA
306 DATA  C9,01     :REM CMP #$01
307 DATA  F0,03     :REM BEQ $C014
308 DATA  4C,ED,F5  :REM JMP $F5ED
309 DATA  A2,00     :REM LDX #$00
310 DATA  BD,3E,C2  :REM LDA $C23E,X
311 DATA  9D,A8,02  :REM STA $02A8,X
312 DATA  E8        :REM INC
313 DATA  E0,48     :REM CPX #$48
314 DATA  D0,F5     :REM BNE $C016
315 DATA  A0,00     :REM LDY #$00
316 DATA  C4,B7     :REM CPY $B7
317 DATA  F0,0A     :REM BEQ $C031
318 DATA  B1,BB     :REM LDA ($BB),Y
319 DATA  99,83,C1  :REM STA $C183,Y
320 DATA  C8        :REM INY
321 DATA  C0,10     :REM CPY #$10
322 DATA  D0,F2     :REM BNE $C023
323 DATA  C0,10     :REM CPY #$10
324 DATA  F0,09     :REM BEQ $C03E
325 DATA  A9,20     :REM LDA #$20
326 DATA  99,83,C1  :REM STA $C183,Y
327 DATA  C8        :REM INY
328 DATA  4C,31,C0  :REM JMP $C031
329 DATA  A9,01     :REM LDA #$01
330 DATA  AA        :REM TAX
331 DATA  A8        :REM TAY
332 DATA  20,BA,FF  :REM JSR $FFBA
333 DATA  A9,BC     :REM LDA #$BC
334 DATA  A2,83     :REM LDX #$83
335 DATA  A0,C1     :REM LDY #$C1
336 DATA  20,BD,FF  :REM JSR $FFBD
337 DATA  A5,AE     :REM LDA $AE
```

```
338 DATA  48          :REM PHA
339 DATA  A5,AF       :REM LDA $AF
340 DATA  48          :REM PHA
341 DATA  A5,C1       :REM LDA $C1
342 DATA  48          :REM PHA
343 DATA  A5,C2       :REM LDA $C2
344 DATA  48          :REM PHA
345 DATA  A9,A8       :REM LDA #$A8
346 DATA  85,C1       :REM STA $C1
347 DATA  A9,02       :REM LDA #$02
348 DATA  85,C2       :REM STA $C2
349 DATA  A9,04       :REM LDA #$04
350 DATA  85,AE       :REM STA $AE
351 DATA  A9,03       :REM LDA #$03
352 DATA  85,AF       :REM STA $AF
353 DATA  A9,D3       :REM LDA #$D3
354 DATA  8D,02,03    :REM STA $0302
355 DATA  A9,02       :REM LDA #$02
356 DATA  8D,03,03    :REM STA $0303
357 DATA  A9,00       :REM LDA #$00
358 DATA  85,9D       :REM STA $9D
359 DATA  20,ED,F5    :REM JSR $F5ED
360 DATA  68          :REM PLA
361 DATA  85,AD       :REM STA $AD
362 DATA  EA          :REM NOP
363 DATA  EA          :REM NOP
364 DATA  85,C2       :REM STA $C2
365 DATA  68          :REM PLA
366 DATA  85,AC       :REM STA $AC
367 DATA  EA          :REM NOP
368 DATA  EA          :REM NOP
369 DATA  85,C1       :REM STA $C1
370 DATA  68          :REM PLA
371 DATA  85,AF       :REM STA $AF
372 DATA  68          :REM PLA
373 DATA  85,AE       :REM STA $AE
374 DATA  20,53,E4    :REM JSR $E453
375 DATA  78          :REM SEI
376 DATA  AD,11,D0    :REM LDA $D011
377 DATA  29,EF       :REM AND #$EF
378 DATA  8D,11,D0    :REM STA $D011
379 DATA  A9,44       :REM LDA #$44
380 DATA  8D,14,03    :REM STA $0314
381 DATA  A9,C1       :REM LDA #$C1
382 DATA  8D,15,03    :REM STA $0315
383 DATA  A5,01       :REM LDA $01
384 DATA  48          :REM PHA
385 DATA  A9,06       :REM LDA #$06
386 DATA  85,01       :REM STA $01
387 DATA  85,C0       :REM STA $C0
388 DATA  A9,03       :REM LDA #$03
389 DATA  A0,00       :REM LDY #$00
390 DATA  84,9B       :REM STY $9B
391 DATA  CA          :REM DEX
392 DATA  D0,FD       :REM BNE $C0B4
393 DATA  88          :REM DEY
394 DATA  D0,FA       :REM BNE $C0B4
395 DATA  38          :REM SEC
396 DATA  E9,01       :REM SBC #$01
397 DATA  D0,F5       :REM BNE $C0B4
```

```
398 DATA  A9,7F     :REM LDA #$7F
399 DATA  8D,0D,DC :REM STA $DC0D
400 DATA  A9,82     :REM LDA #$82
401 DATA  8D,0D,DC :REM STA $DC0D
402 DATA  A9,47     :REM LDA #$47
403 DATA  8D,06,DC :REM STA $DC06
404 DATA  E8        :REM INX
405 DATA  8E,07,DC :REM STX $DC07
406 DATA  A9,07     :REM LDA #$07D
407 DATA  85,B4     :REM STA $B47
408 DATA  A9,19     :REM LDA #$196
409 DATA  8D,0F,DC :REM STA $DC0F
410 DATA  AD,0D,DC :REM LDA $DC0D
411 DATA  58        :REM CLI #$07
412 DATA  A9,03     :REM LDA #$03
413 DATA  20,7A,C1 :REM JSR $C17A
414 DATA  88        :REM DEY
415 DATA  D0,F8     :REM BNE $C0DF
416 DATA  A9,48     :REM LDA #$48
417 DATA  20,7A,C1 :REM JSR $C17A
418 DATA  A5,C1     :REM LDA $C1
419 DATA  20,7A,C1 :REM JSR $C17A
420 DATA  A5,C2     :REM LDA $C2
421 DATA  20,7A,C1 :REM JSR $C17A
422 DATA  A5,AE     :REM LDA $AE
423 DATA  38        :REM SEC
424 DATA E5,AC      :REM SBC $AC
425 DATA 48         :REM PHA
426 DATA A5,AF      :REM  LDA $AF
427 DATA E5,AD      :REM SBC $AD
428 DATA AA         :REM TAX
429 DATA 68         :REM PLA
430 DATA 18         :REM CLC
431 DATA 65,C1      :REM ADC $C1
432 DATA 20,7A,C1   :REM JSR $C17A
433 DATA 8A         :REM TXA
434 DATA 65,C2      :REM ADC $C2
435 DATA 20,7A,C1   :REM JSR $C17A
436 DATA B1,AC      :REM LDA ($AC),Y
437 DATA AA         :REM TAX
438 DATA 45,9B      :REM EOR $9B
439 DATA 85,9B      :REM STA $9B
440 DATA 8A         :REM TXA
441 DATA 20,7A,C1   :REM JSR $C17A
442 DATA 20,DB,FC   :REM JSR $FCDB
443 DATA 20,D1,FC   :REM JSR $FCD1
444 DATA D0,ED      :REM BNE $C10E
445 DATA A5,9B      :REM LDA $9B
446 DATA 20,7A,C1  :REM  JSR $C17A
447 DATA 20,7A,C1  :REM  JSR $C17A
448 DATA 20,7A,C1  :REM  JSR $C17A
449 DATA 78        :REM  SEI
450 DATA 68        :REM  PLA
451 DATA 09,20     :REM  ORA #$20
452 DATA 85,01     :REM  STA $01
453 DATA 20,15,FD :REM  JSR $FD15
454 DATA 20,A3,FD :REM  JSR $FDA3
455 DATA 20,93,FC :REM  JSR $FC93
456 DATA A9,0E     :REM  LDA #$0E
457 DATA 8D,20,D0 :REM  STA $D020
458 DATA 58        :REM  CLI
```

```
459 DATA 4C,74,A4 :REM  JMP $A474
460 DATA 06,BF    :REM  ASL $BF
461 DATA 90,05    :REM  BCC $C14D
462 DATA A2,33    :REM  LDX #$33
463 DATA CA       :REM  DEX
464 DATA D0,FD    :REM  BNE $C14A
465 DATA A5,01    :REM  LDA $01
466 DATA 29,F7    :REM  AND #$F7
467 DATA 85,01    :REM  STA $01
468 DATA A2,19    :REM  LDX #$19
469 DATA 8E,0F,DC :REM  STX $DC0F
470 DATA A2,10    :REM  LDX #$10
471 DATA CA       :REM  DEX
472 DATA D0,FD    :REM  BNE $C15A
473 DATA 09,08    :REM  ORA #$08
474 DATA 85,01    :REM  STA $01
475 DATA EE,20,D0 :REM  INC $D020
476 DATA C6,B4    :REM  DEC $B4
477 DATA 10,0C    :REM  BPL $C174
478 DATA A9,07    :REM  LDA #$07
479 DATA 85,B4    :REM  STA $B4
480 DATA A5,BE    :REM  LDA $BE
481 DATA 85,BF    :REM  STA $BF
482 DATA A9,80    :REM  LDA #$80
483 DATA 85,9C    :REM  STA $9C
484 DATA AD,0D,DC :REM  LDA $DC0D
485 DATA 4C,BC,FE :REM  JMP $FEBC
486 DATA 24,9C    :REM  BIT $9C
487 DATA 10,FC    :REM  BPL $C17A
488 DATA 85,BE    :REM  STA $BE
489 DATA 84,9C    :REM  STY $9C
490 DATA 60       :REM  RTS
491 DATA 20,20,20 :REM  SZOKOZOK
492 DATA 20,20,20 :REM
493 DATA 20,20,20 :REM
494 DATA 20,20,20 :REM
495 DATA 20,20,20 :REM
496 DATA 20       :REM
497 DATA 78       :REM  SEI
498 DATA A9,0B    :REM  LDA #$0B
499 DATA 8D,11,D0 :REM  STA $D011
500 DATA A9,A8    :REM  LDA #$A8
501 DATA 8D,14,03 :REM  STA $0314
502 DATA A9,02    :REM  LDA #$02
503 DATA 8D,15,03 :REM  STA $0315
504 DATA A5,01    :REM  LDA $01
505 DATA 29,1F    :REM  AND #$1F
506 DATA 85,01    :REM  STA $01
507 DATA 85,C0    :REM  STA $C0
508 DATA A0,00    :REM  LDY #$00
509 DATA 84,9B    :REM  STY $9B
510 DATA CA       :REM  DEX
511 DATA D0,FD    :REM  BNE $C1AF
512 DATA 88       :REM  DEY
513 DATA D0,FA    :REM  BNE $C1AF
514 DATA A9,7F    :REM  LDA #$7F
515 DATA 8D,0D,DC :REM  STA $DC0D
516 DATA A9,90    :REM  LDA #$90
517 DATA 8D,0D,DC :REM  STA $DC0D
518 DATA A9,FE    :REM  LDA #$FE
```

```
519 DATA 8D,06,DC :REM  STA $DC06
520 DATA A9,01    :REM  LDA #$01
521 DATA 8D,07,DC :REM  STA $DC07
522 DATA A9,19    :REM  LDA #$19
523 DATA 8D,0F,DC :REM  STA $DC0F
524 DATA AD,0D,DC :REM  LDA $DC0D
525 DATA 58       :REM  CLI
526 DATA A5,BF    :REM  LDA $BF
527 DATA C9,03    :REM  CMP #$03
528 DATA D0,FA    :REM  BNE $C1D2
529 DATA A9,07    :REM  LDA #$07
530 DATA 85,B4    :REM  STA $B4
531 DATA A2,10    :REM  LDX #$10
532 DATA 86,9C    :REM  STX $9C
533 DATA 20,CA,02 :REM  JSR $02CA
534 DATA C9,03    :REM  CMP #$03
535 DATA D0,EB    :REM  BNE $C1D2
536 DATA CA       :REM  DEX
537 DATA D0,F6    :REM  BNE $C1E0
538 DATA 20,CA,02 :REM  JSR $02CA
539 DATA C9,03    :REM  CMP #$03
540 DATA F0,F9    :REM  BEQ $C1EA
541 DATA C9,48    :REM  CMP #$48
542 DATA D0,DD    :REM  BNE $C1D2
543 DATA 20,CA,02 :REM  JSR $02CA
544 DATA 95,AC    :REM  STA $AC,X
545 DATA E8       :REM  INX
546 DATA E0,04    :REM  CPX #$04
547 DATA D0,F6    :REM  BNE $C1F5
548 DATA 20,CA,02 :REM  JSR $02CA
549 DATA 91,AC    :REM  STA ($AC),Y
550 DATA 45,9B    :REM  EOR $9B
551 DATA 85,9B    :REM  STA $9B
552 DATA 20,DB,FC :REM  JSR $FCDB
553 DATA 20,D1,FC :REM  JSR $FCD1
554 DATA D0,EF    :REM  BNE $C1FF
555 DATA 20,CA,02 :REM  JSR $02CA
556 DATA C5,9B    :REM  CMP $9B
557 DATA F0,03    :REM  BEQ $C21A
558 DATA 4C,E2,FC :REM  JMP $FCE2
559 DATA 78       :REM  SEI
560 DATA 20,15,FD :REM  JSR $FD15
561 DATA 20,A3,FD :REM  JSR $FDA3
562 DATA 20,93,FC :REM  JSR $FC93
563 DATA A9,0E    :REM  LDA #$0E
564 DATA 8D,20,D0 :REM  STA $D020
565 DATA A6,AE    :REM  LDX $AE
566 DATA A4,AF    :REM  LDY $AF
567 DATA 58       :REM  CLI
568 DATA 4C,D9,02 :REM  JMP $02D9
569 DATA A9,E1    :REM  LDA #$E1
570 DATA 8D,28,03 :REM  STA $0328
571 DATA A9,00    :REM  LDA #$00
572 DATA 20,71,A8 :REM  JSR $A871
573 DATA 4C,AE,A7 :REM  JMP $A7AE
574 DATA AD,0D,DC :REM  LDA $DC0D
575 DATA A2,19    :REM  LDX #$19
576 DATA 8E,0F,DC :REM  STX $DC0F
577 DATA 4A       :REM  LSR
578 DATA 4A       :REM  LSR
```

```
579 DATA 26,BF      :REM  ROL $BF
580 DATA EE,20,D0 :REM  INC $D020
581 DATA C6,B4      :REM  DEC $B4
582 DATA 10,0C      :REM  BPL $C25D
583 DATA A9,07      :REM  LDA #$07
584 DATA A9,07      :REM  LDA #$07
585 DATA 85,B4      :REM  STA $B4
586 DATA A5,BF      :REM  LDA $BF
587 DATA 85,BE      :REM  STA $BE
588 DATA A9,80      :REM  LDA #$80
589 DATA 85,9C      :REM  STA $9C
590 DATA 4C,BC,FE :REM  JMP $FEBC
591 DATA 24,9C      :REM  BIT $9C
592 DATA 10,FC      :REM  BPL $C260
593 DATA 84,9C      :REM  STY $9C
594 DATA A5,BE      :REM  LDA $BE
595 DATA 60         :REM  RTS
596 DATA 20,53,E4 :REM  JSR $E453
597 DATA 4C,51,03 :REM  JMP $0351
598 DATA 86,2D      :REM  STX $2D
599 DATA 84,2E      :REM  STY $2E
600 DATA 20,59,A6 :REM  JSR $A659
601 DATA 20,33,A5 :REM  JSR $A533
602 DATA 4C,EF,03 :REM  JMP $03EF
603 DATA A4,A4      :REM  KODOK
604 DATA A4,A4      :REM
605 DATA A4,A4      :REM
606 DATA A4,A4      :REM
607 DATA A4         :REM
```

5.5.4 Az operációs rendszer hivatkozott címei és rutinjai

KMR programban kár lenne nem felhasználni az operációs rendszer bizonyos rutinjai adta lehetőségeket. Ezek a címek és rutinok más különböző védelmi rendszerek fejlesztése közben is alkalmazhatók. Éppen ezért a következőkben a legfontosabbakat dokumentáljuk.

25. táblázat

Nulláslap címek

Tárcím		Jelentés
hex.	dec.	
$2B/$2C	43/44	A BASIC program kezdete
$2D/$2E	45/46	A BASIC tár vége
$2F/$30	47/48	Mutató a tömbök elejére
$31/$32	49/50	Mutató a szabad RAM elejére (szövegek)
$3D/$3E	61/62	Mutató az aktuális BASIC sor kezdetére
$90	144	Státusz (a BASIC ST változó)
$93	147	LOAD($00)/VERIFY($01) kapcsoló

25. táblázat folytatása

Tárcím		Jelentés
hex.	dec.	
$90	157	parancs ($80), program ($00) üzemmódkapcsoló ($C0 esetén minden hibaüzenet megjelenik, az operációs rendszer üzenetei is; $80 esetén csak a BASIC értelmező hibaüzenetei; $00 esetén a hibaüzenetek egyáltalán nem jelennek meg)
$A6	166	A kazettapufferbe kiírt, ill. onnan beolvasott byte-ok száma
$AC/$AD	172/173	LOAD/SAVE futtatáscím
$AE/$AF	174/175	Mutató a *végcím* + *1*-re, LOAD/SAVE-hez
$B2/$B3	178/179	A kazettapuffer elejének mutatója
$B7	183	A file-név hossza
$B8	184	Logikai file-szám
$B9	185	Másodlagos cím
$BA	186	Egységszám
$BB/$BC	187/188	Mutató a file-névre
$C0	192	Ha a kazetta billentyűje le van nyomva és a tárcím tartalma nem nulla, a motor nem kapcsol be
$C1/$C2	193/194	A beolvasás/kiírás kezdőcíme
$C3/$C4	195/196	A beolvasás/kiírás végcíme
$C6	198	Billentyűzetpuffer karaktereinek száma
$D3	211	Aktuális kurzorpozíció a soron belül (oszlop)
$D6	214	Aktuális kurzorsor

Az operációs rendszer és a BASIC értelmező rutinjai:

$A000	Bekapcsolás utáni állapot felvétele
$A474	Ugrás a BASIC értelmező beolvasó ciklusára
$A533	Programsorok újraláncolása
$A659	A CLR utasítás
$A7AE	A BASIC értelmező utasítást végrehajtó ciklusa
$A871	A RUN parancs
$AAA0	A PRINT utasítás
$E17A	Betöltési hiba ellenőrzése, hibaüzenet a $A1A1 értéke alapján
$E1D4	A file-név, a másodlagos cím és az egységszám beolvasása a BASIC szövegből
$EE453	A BASIC értelmező összes vektorának inicializálása
$F68F	A SAVING"file-név" szöveg kiírása

$F750	A FOUND"file-név" szöveg kiírása és várakozás a "C=" billentyűre
$F80D	A puffermutató növelése $A6-ra
$F817	Várakozás a kazettagomb lenyomására, és a "PRESS RECORD AND PLAY ON TAPE" szöveg kiírása
$FC93	A felvétel üzemmód kikapcsolása
$FCD1	A $AE/$AF és $AC/$AD címek tartalmának összehasonlítása
$FCDB	$AC/$AD növelése
$FCE2	RESET, a számítógép alaphelyzetbe kerül
$FD50	RAM teszt és inicializálás
$FDAE	A CIA-k inicializálása
$FEBC	Az IRQ rutin befejezése
$FF8A	Az operációs rendszer vektorainak visszaállítása
$FFBA	FILPAR, beállítja az egységszám, a másodlagos cím és file-név paramétereket
$FFBD	FILNAM, beállítja a file-név paramétereket
$FFC0	OPEN
$FFC3	CLOSE
$FFC6	CHKIN, a beolvasóegység kijelölése
$FFC9	CHKOUT, a kiíróegység kijelölése
$FFCC	CLRCH, beolvasás, kiírás után alaphelyzetbe állítás
$FFCF	BASIN, egy karakter beolvasása
$FFD2	BASOUT, egy karakter kiírása
$FFD5	LOAD (vagy VERIFY)
$FFD8	SAVE
$FFE7	CLALL, az összes nyitott csatorna lezárása

A könyv programvédelmi
és másolásvédelmi
eljárásokat tárgyal.
Nemcsak összefoglalja
az eddig alkalmazott
legismertebb eljárásokat,
hanem bemutatja
az új módszereket is.
A C 64-es és C 128-as gépekre
teljes megoldásokat tárgyal,

DR. LENGYEL JÓZSEF - VARGA ANTAL - LAKAT ALATT
VÉDELMI MÓDSZEREK C64-ESRE

érthetően dokumentálva
(programlisták, gépi nyelvű listák).
Kitér a kazetta és mágneslemez
másolásának védelmére,
bemutat védett programokat
assembly és BASIC nyelven.

NOVOTRADE

280,– FT