Liesert

PEEK-ek

és

POKE-ok

a C 64-esen

DATA BECKER–NOVOTRADE

Liesert

PEEK-ek

és

POKE-ok

a C 64-esen

DATA BECKER–NOVOTRADE

A könyv eredeti címe: PEEKS & POKES zum Commodore 64 (1986)

Fordította: APEX Szervező, Szolgáltató GMK

Lektorálta: DR. LENGYEL JÓZSEF
DR. SZIDAROVSZKI FERENC

A kiadásért felel RÉNYI GÁBOR, a NOVOTRADE RT. igazgatója
Budapest, 1987

Készült a Somogy Megyei Nyomdaipari Vállalat kaposvári üzemében

Felelős vezető: MIKE FERENC igazgató

ISBN 963 02 5068 3

FONTOS TUDNIVALÓK

A könyvben ismertetett kapcsolások, eljárások és programok nem tekinthetők szabadalmi oltalom alá eső ipari termékeknek. Ezek elsősorban amatőr és oktatási célokat szolgálnak. A szerzők rendkívül nagy gondot fordítottak a kapcsolások, műszaki adatok és programok helyességére, a részletek kidolgozása során többszöri ellenőrzést végeztek. Mindez azonban nem zárja ki az esetleges hibalehetőségeket. Az előforduló hibákért és az ebből adódó következményekért a DATA BECKER cég sem szavatosságot, sem jogi felelősséget nem vállal. Az esetlegesen előforduló hibák közlését a szerzők hálásan fogadják.

TARTALOMJEGYZÉK

ELŐSZÓ

A probléma ismerős: átolvastuk a C 64-eshez mellékelt használati utasítást és alig telik el egy kis idő, máris többre vagyunk kíváncsiak, mint amennyit ebből megtudhatunk. Olyan kérdések merülnek fel például, hogy hogyan kell elvégezni a sprite-ok ütközésvizsgálatát, hogyan állíthatunk elő nagyfelbontású grafikát vagy hogyan kérdezhetünk le egyszerre két billentyűt. A függelékben megtaláljuk ugyan a nulláslapra vonatkozó jegyzéket, de:

1. mi is tulajdonképpen az a *nulláslap* és
2. hogyan használhatjuk?

Ha ezekre a kérdésekre szeretnénk választ kapni, akkor most éppen a legmegfelelőbb könyvet tartjuk a kezünkben.

Gondolatban egy kis utazást teszünk majd a C 64-es tárolójában és operációs rendszerében. Hogy még azt sem tudjuk, mi az az "operációs rendszer"? Nem tesz semmit, időközben erre is fény derül.

A könyv három részből áll. Az elsőben azokat az alap dolgokat tárgyaljuk meg, amelyek ismerete feltétlenül szükséges a második részben bemutatott trükkök megértéséhez. Ide tartoznak a PEEK és POKE utasítások is. Ha ezek után már kellően ismerjük a gépet, tudjuk, hogy működik és miként illetve mire programozható, következik egy csomó trükk, ami kivétel nélkül mind BASIC-ben működik. Megtanulunk tehát a gépi nyelv ismerete nélkül kényelmesen programozni. A trükköket ismertető fejezetek végén rövid összefoglalót találunk, hogy a későbbiek során ne kelljen mindig az összes magyarázatot újra végigolvasni. Aki az előírásokat pontosan betartja, biztos lehet benne, hogy minden trükk és program működni fog.

Néhány szó a gépi nyelvről: A 13. fejezetben egy mikroprocesszor szimulátorprogram található és ennek kapcsán rövid bevezetést nyújtunk a gépi nyelvű programozásba.

A VC 20 tulajdonosoknak azt mondhatjuk, hogy minden, amit itt a nulláslappal kapcsolatban elmondunk, elvileg az ő gépükre is érvényes, még ha némi módosítással is. Érdemes átnézni erre vonatkozóan néhány más DATA BECKER kiadványt.

E könyv szerzőjének nincs már más hátra, mint sok sikeret kívánni a CBM programozás új lehetőségeinek kiaknázásához.

Münster, 1984. május — Hans Joachim Liesert

1. A SZÁMÍTÓGÉP MŰKÖDÉSE

A következő fejezetben a C 64-es működésével ismerkedünk meg. Akik a számítógép technikában már valamelyest járatosak, ezt a részt nyugodtan átlapozhatják, a szakemberektől pedig elnézést kérünk, amiért bizonyos dolgokat, a könnyebb érthetőség kedvéért, leegyszerűsítve tárgyalunk.

1.1. Egy 16 lábú "pók": a mikroprocesszor

Mindenekelőtt egy kevés alapismeret. A mikroprocesszorok feladata, hogy a számítógép tárolótartományát megcímezzék, azaz aktivizálják egy-egy meghatározott számú (című) tárolórekeszét. Az, hogy egy mikroprocesszor hány tárolórekesz címzésére képes, attól függ, hogy hány címvezetéke (lába) van. Minden címvezeték egy bitet képvisel és mint ilyen, összesen két állapotot vehet fel: 0 (= áram nincs) és 1 (= áram van).

A 64-es "ágya" egy 16 címvezetékű 6510-es mikroprocesszor. Ezzel a címbusszal (= a címvezetékek összessége) összesen $2^{16} = 65536$ tárolórekeszt képes elérni.

Minden egyes tárolórekesz egy 8 bites (azaz 1 byteos) információ tárolására alkalmas. Amikor a 6510-es mikroprocesszor a 16 lábával (vagyis címvezetékével) kiválasztott és leírt egy tárolórekeszt, akkor a 8 kezével (ill. adatvezetékével) információkat helyezhet el benne vagy vehet ki belőle.

Mivel az adatbusz szélessége éppen fele a címbusz szélességének, egy-egy cím ábrázolásához két byte-ra van szükség.

A mikroprocesszornak saját, külön számrendszere van, a tizenhatos (vagy hexadecimális) számrendszer. Ebben, mint a neve is mutatja, a számok ábrázolásához összesen 16 "számjegy" áll rendelkezésünkre. Számok: 0–9 és betűk: A–F.

Ennek az az igen nagy előnye, hogy bármelyik hexadecimális számjegy kifejezhető négy bittel (pl.: F=15=1111), azaz négy bit éppen egy hexaszámjegyet ad.

Egy byte ábrázolásához tehát 2, egy címhez 4 hexadecimális számjegy szükséges. Ez minden 8-bites processzorra érvényes, ahol a 8 bit az adatbusz szélességét jelenti. Ezek után térjünk rá a 64-es speciális tulajdonságaira.

1.2. Mi is az az operációs rendszer?

A számítógéptechnika irodalmában gyakran találkozunk olyan kifejezésekkel, mint "operációs rendszer", "interpreter" vagy (különösen a 64-essel kapcsolatban) "interrupt". Nos, az interpreter nem más, mint egy értelmező. A továbbiakban mi is ezt az elnevezést fogjuk használni. Feladata már a nevéből is sejthető: értelmezi az általunk BASIC-ben beadott utasítást, azaz átalakítja a mikroprocesszor által "érthető", feldolgozható formában.

A 64-es és minden más személyi számítógép kizárólag a saját gépi kódú nyelvét érti. Csak a ROM-ban tárolt BASIC-értelmező képes arra, hogy a RAM-ban levő programsorokat kiolvassa és feldolgozza. Amikor közvetlen üzemmódban dolgozunk, az értelmező nem a tárolóból veszi az utasításokat, hanem a BASIC-beviteli pufferből. Ez egy "átadás-átvételi" tároló a billentyűzetről történő bevitelek számára.

Az operációs rendszer (ami szintén egy ROM-ban tárolt program) lekérdezi a billentyűzetet, létrehozza a kurzort és kezeli a perifériákat.

Az operációs rendszer is és az értelmező is gépi kódú programok. Mindkettő a számítógép bekapcsolásával együtt indul és mindaddig fut, míg egy másik gépi kódú programot el nem indítunk. Ez a SYS-utasítással lehetséges. Az ilyen gépi kódú programok feldolgozása után a számítógép visszatér a BASIC-be.

Azt tudjuk, hogy egy gépen egyszerre csak egy program dolgozható fel, kivéve az ún. multiprocesszoros rendszereket. Az értelmező és az operációs rendszer viszont két különböző program, amelyeknek bizonyos feladatok elvégzéséhez egyszerre kell futniuk. Hogy oldható fel ez az ellentmondás?

Annak, hogy két program megközelítően egyidőben fusson, legegyszerűbb módja, ha egymást kölcsönösen hívják. Amikor például a BASIC elvégzi munkájának egy részét, átkapcsol az operációs rendszerre, és fordítva. Ez történik például, amikor valamelyik perifériát kell működtetni. A BASIC csupán az információt bocsátja rendelkezésre, amit azután az operációs rendszer továbbít a készülék felé. A billentyűzet is csak akkor kérdezhető le, ha az operációs rendszer fut. Ismeretes viszont, hogy a feldolgozás alatt lévő BASIC programot a RUN/STOP billentyűvel megszakíthatjuk. A megfelelő hatás eléréséhez tehát alkalmat kell teremteni ennek lekérdezésére. A számítógép gyártók erre találták ki az ún. interrupt-ot, magyarul megszakítást. Az éppen futó gépi nyelvű programot (akár BASIC vagy operációs rendszer) a processzor minden 1/60 másodpercben megszakítja és a billentyűzet lekérdező alprogramba ugrik. (Eközben még néhány más dolgot is végrehajt.) Ha a számítógép "észreveszi", hogy lenyomtuk a RUN/STOP billentyűt, az

éppen futó BASIC programot megszakítja. Más billentyűk lenyomását is tudomásul veszi és tárolja őket a billentyűzetpufferban. A felhasználónak úgy tűnik, mintha a billentyűzet lekérdezése folyamatos lenne, hiszen még a legügyesebb gépíró sem tudna másodpercenként több, mint 15 jelet bevinni. A mikroprocesszor számára viszont a két megszakítás közt eltelt idő egy örökkévalóság, hiszen másodpercenként kb. 980 000 műveletet képes végrehajtani és egy-egy gépi kódú utasítás végrehajtásához átlagban 3–4 ilyen művelet szükséges. A megszakításig tehát többezer utasítást képes feldolgozni. A billentyűzet lekérdezése után a program feldolgozása ott folytatódik, ahol a megszakításkor abbamaradt.

Sajnos a megszakító rutinnak van egy nemkívánatos mellékhatása. Minden lefutáskor módosítja ugyanis – számunkra esetleg kedvezőtlenül – a tároló meghatározott byte-jait. Emiatt nem férhetünk hozzá minden további nélkül ahhoz a RAM-tartományhoz, amely a ROM-mal átfedésben van – de erről majd később.

Végül tegyünk egy kísérletet. Az alábbiakban közöljük egy FOR . . . NEXT ciklust tartalmazó program listáját. Ennek feldolgozásához a gépnek kb. 46 másodpercre van szüksége.

A megszakító rutint kikapcsolva (pontos működésére egy későbbi fejezetben még visszatérünk) mindez egy teljes másodperccel gyorsabban fut le! Ez a nagyobb sebesség azonban nem mérhető a CMB-BASIC TI$ változójával, mert a megszakító rutin a belső óra továbbállítását is végrehajtja.

Mielőtt a programot begépeljük, próbáljuk ki közvetlen üzemmódban a 10. sor POKE utasítását! Ha eltűnik a kurzor és a billentyűzet sem kérdezhető le, kikapcsoltuk a megszakító rutint – már csak a RUN/STOP–RESTORE segíthet. Jó szórakozást!

```
1 REM *** P 1 ***
2 REM
10 POKE56334,PEEK(56334)AND254:REM MEGSZAKITAS KI
20 FOR I=1 TO 1000:PRINT I:NEXT I
30 POKE56334,PEEK(56334)OR1:REM MEGSZAKITAS BE
READY.
```

1.3. Hogyan működik a BASIC értelmező?

Mint már mondtuk, a BASIC értelmező feladata a BASIC utasítások feldolgozása. A felhasználó számára, aki ennek csak az eredményét, a programfutást látja, érdekes lehet magának az értelmezőnek a működése. Kezdjük az utasítások bevitelével. A 64-es a BASIC sorokat nem egyszerűen betűk sorozataként tárolja, mert ez túl sok tárolóhelyet igényelne. Egy egyszerű PRINT utasítás például 5 byte-ot (betűnként 1-et). Minden utasításszó ún. *token*ként, az ASCII-hoz hasonló értelmezői kódként kerül rögzítésre.

Azok a számok és betűk, amelyek nem képeznek utasításszót, ASCII-kódjukkal kerülnek a tárolóba. Így a PRINT I utasítás mindössze 2 byte tárolókapacitást igényel, egyet az utasításszó, egyet pedig a változónév számára. Ezzel a fordítás első része kész is és bármilyen hihetetlen, a valóságban mindez a RETURN lenyomása és a kurzor újbóli megjelenése közt eltelt igen rövid idő alatt zajlik le. Gyakran még az is előfordul, hogy eközben az egész programszöveget el kell tolni a tárolóban, pl. új sor közbeiktatásakor.

A fordítás második része RUN bevitelét követően kezdődik. A tokenek alapján az értelmező a megfelelő alprogramjaiba ugrik, amelyek ezután átveszik a tulajdonképpeni munkát. A PRINT utasításhoz pl. olyan alprogramok tartoznak, mint a "kifejezés kiértékelése" (a képernyőn megjelenítendő változó) vagy a "karakter kivitele a képernyőre", ami minden egyes jel kivitelekor lefut. Más utasítások számára további alprogramok vannak a ROM-ban, így pl. matematikai rutinok és hasonlók.

Természetesen itt most minden rutint részletesen elemezhetnénk, de ez már meghaladná könyvünk kereteit. Akit ezek a dolgok bővebben is érdekelnek, vegye elő az "A Commodore 64-es belső felépítése" c. DATA BECKER–NOVOTRADE kiadványt.

Ez tartalmazza egyebek között a BASIC értelmező és az operációs rendszer teljes ROM listáját és sok hasznos gépi kódú programot is.

1.4. PEEK, POKE és egyebek

Képzeljük el a következőt: valamelyik szaklapban egy bevitelre kész szuper-programra bukkanunk. Máris begépeljük a mintegy 20 k-nyi listát, elindítjuk a programot, ám az idő előtt egy hibaüzenettel megszakad. Nincs mese, meg kell értenünk a program működését, ha csak nem akarjuk az egész listát

betűről betűre összehasonlítani az eredetivel. Csak ne volna ez a rengeteg POKE – utasítás! Normális programozó nem is használ ilyet! Éppen itt az ideje tehát, hogy megfejtsük ezeknek az utasításoknak a titkát.

1.4.1. PEEK és POKE

Nézzük először a POKE utasítást. Írásmódja már ismert:

POKE cím, byte

A cím értéke 0 és 65535, a byte-é 0 és 255 között lehet. Az utasítás feladata, hogy egy byte-ot beírjon a megadott című tárolórekeszbe. Célja több is lehet: meghatározhatjuk a színeket, a címtől függően írhatunk a képtárolóba stb. Ezzel alaposan belekontárkodhatunk a számítógép dolgába, hiszen az operációs rendszer és az értelmező is kénytelen tárolni bizonyos adatokat. Hogy ezek mik, azt a PEEK utasítással tudhatjuk meg. Ez az utasítás éppen a POKE fordítottja, vagyis a tároló tartalmának kiolvasására alkalmas. Írásmódja szintén ismerős:

PEEK cím

A PRINT PEEK cím utasítás kiolvassa és kiírja az adott címen tárolt byte-ot.

Fontos megjegyezni, hogy a PEEK utasítás függvénytípusú, ezért csak hozzárendelésként (pl.: A=PEEK...) vagy egyéb kifejezésekben használható.

A két közös utasítás közös jellemzője, hogy célját az adott cím határozza meg. Emiatt mindig ajánlatos a tár foglaltsági térképéből megállapítani azt, hogy éppen melyik tárterületen dolgozunk. Ebből azután az utasítás célja, feladata már megállapítható.

1.4.2. SYS és USR

Ez a két utasítás tulajdonképpen csak azok számára érdekes, akik gépi nyelven programoznak. Írásmódjuk:

SYS cím és
PRINT USR (X)

Mindkettő egy gépi kódú program lehívására alkalmas. A SYS-t követő cím annak a tárolórekesznek a száma, ahol a lehívott gépi nyelvű program kezdődik. Ennek végrehajtása után az értelmező visszatér a BASIC-főprogramba.

Az USR függvény hasonlóképpen működik, de további hasznos feladata is van. Mivel függvényként használjuk, eltérő az írásmódja. A PEEK-hez hasonlóan csak valamilyen kifejezésben szerepelhet (ld. előző pont). További eltérés, hogy a gépi kódú program kezdőcímét nem kell megadnunk. Ezt egy "elektronikus postaláda", a 785-ös és 786-os tárolórekesz tartalmazza. Amikor az értelmező egy USR utasításra bukkan, kiolvassa e két rekesz tartalmát és az így megadott tárolócímre ugorva végrehajtja a gépi nyelvű programot.

Ezután visszatér BASIC-be.

A legfontosabb, hogy ezzel az utasítással adatokat adhatunk át a gépi kódú programnak, és fordítva. A zárójelben szereplő értéket az értelmező elhelyezi az ún. lebegőpontos akkumulátorban (97–101). Ez az a belső regiszter, ahol minden aritmetikai művelet végrehajtásra kerül. A gépi kódú program ezt az értéket kiolvassa és feldolgozza. Az USR-rutin végén a gép a lebegőpontos akkumulátorban levő értéket átadja a BASIC-nek. Ezzel az eljárással a változóknak a gépi program adhat értéket (pl. A=USR (X)).

Az utasítás rendeltetése az, hogy a programozó bizonyos dolgokra (pl. válogatás, rendezés) saját gépi kódú rutinokat írhasson és azokat BASIC-ből futtathassa.

Ebből a szempontból az USR egy "szuperutasítás".

1.4.3. Egy kis kitérő a bináris aritmetikába

Az eddig leírtakhoz csatlakozik még néhány utasítás, amelyek ismertek ugyan, de valószínűleg nem közismert a sokoldalúságuk. Mindenekelőtt itt van az AND, az OR és a NOT. Ezidáig csak IF ... THEN szerkezetekben találkoztunk velük, mint pl.:

```
IF A=0  AND B=0  THEN 100,
```

pedig ezeket eredetileg számok és változók logikai összekapcsolására találták ki. Ehhez tudnunk kell, hogy a számítógép az összehasonlításokat is számokként kezeli. Próbáljuk ki a következő utasításokkal:

```
PRINT (1=2) és
PRINT (1=1)
```

Ha az összeállítás "igaz", az eredmény –1, ha "hamis", akkor 0.

A bináris rendszerben a –1 alakja: 1111 1111. Ha a balszélső bitet nem előjelként értelmezzük, ugyanez az alak a decimális 255-öt jelenti. Kérdés, hogy mit kezdjen ezzel egy BASIC-utasítás.

A program mindig akkor lép ki egy IF ...THEN szerkezetből, ha az összehasonlítás eredménye 0. Így a következő utasítás-sorozat is elképzelhető: IF 3 * A THEN 110. A gép az egyes összehasonlítások eredményeit egyszerűen összekapcsolja és az így kapott végeredmény határozza meg a további programfutást. Az összekapcsolások megértése céljából egy kis kirándulást teszünk a bináris aritmetikába.

Az AND, OR és NOT úgynevezett BOOLE-operátorok, amelyek logikai állapotok összekapcsolására alkalmasak. Ezeket, mint már tudjuk a bitekkel nagyon egyszerűen ábrázolhatjuk (0=hamis, 1=igaz).

Mindig két-két bitet kapcsolunk össze. Az eredmény a következő táblázatok szerint alakul.

AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

Ebből megállapíthatjuk, hogy az összekapcsolás eredménye mindkét esetben 1, ha mindkét bemeneti bit 1. A műveletek szó szerint lefordíthatók, azaz AND-nél az eredmény akkor 1, ha az 1. bit is ÉS a 2. bit is 1, OR-nál pedig akkor, ha az 1. bit VAGY a 2. bit 1.

A NOT-tal más a helyzet. Ez az operátor egyszerűen invertálja a bemeneti bitet.

NOT	0	1
	1	0

Nos, ez eddig rendben van. Sajnos nekünk, egyszerű kis 'Basic-programozóknak', maradt egy problémánk. BASIC-ben ugyanis nem sokra megyünk egyedi bitekkel, hiszen ott decimális számokkal dolgozunk. Ahhoz, hogy megtudjuk mit jelent a 45 AND 123 kifejezés, a következőképpen kell eljárnunk:

1. A decimális számot binárissá alakítjuk

Ez jóval egyszerűbb, mint gondolnánk. A decimális számot folyamatosan el kell osztanunk 2-vel mindaddig, míg 0-t nem kapunk. Az egyes osztások során kapott maradékokat (ami csak 1 vagy 0 lehet) bitként jegyezzük és fordított sorrendben egymás mellé írjuk. Ezzel készen is vagyunk.

Nézzünk egy példát:

```
23:2=11   marad  1
11:2= 5   marad  1
 5:2= 2   marad  1
 2:2= 1   marad  0
 1:2= 0   marad  1
-------------------
          1 0 1 1 1
```

Az eljárás fordítottját a CBM-kézikönyv sprite-okról szóló fejezetéből már ismerjük.

A 45 és a 123 bináris megfelelői a következők:

```
 45=00101101
123=01111011
```

2. A két számot bitenként összekapcsoljuk

Példánkban ez a következőképpen alakul:

```
      00101101
AND   01111011
--------------
      00101001
```

3. Az eredményt decimálissá alakítjuk

00101001=41

Ezt az eredményt sokkal egyszerűbben is megkaphattuk volna, ha begépeljük a PRINT 45 AND 123 utasítást, de akkor magát a folyamatot nem kísérhettük volna figyelemmel.

Jogosan merülhet fel a kérdés, hogy mire jó ez az egész. Nos, az eddig elmondottakon túl ez az eljárás az egyes bitek befolyásolására is alkalmas. Az AND 254 kapcsolat például a jobb szélső bit törlését, míg az OR 1 annak bekapcsolását jelenti. Próbáljuk ki tetszőleges számokkal!

Ezek után maradt még egy titokzatos utasításunk, a

WAIT.

Írásmódja:

WAIT cím, **x**, **y**

Feladata a várakozás, ami a byte-ok sorozatos összekapcsolásából adódik. Ez a következőképpen zajlik le:

Amikor az értelmező egy WAIT – utasítást talál, kiolvassa a megadott című tárolórekesz tartalmát és közte, valamint az **y** szám között kizáró – VAGY (exkluzív-vagy) kapcsolatot hoz létre. Ez, mint a nevéből sejthető, az OR (=VAGY) utasítás rokona, rövidítése XOR. A következőképpen működik:

XOR	0	1
0	0	1
1	1	0

Az eredmény akkor 1, ha csak és kizárólag az egyik bemeneti bit 1. A XOR kapcsolat eredménye és az X szám között ezután egy AND kapcsolatot létesít és az egész folyamat addig ismétlődik, míg a végeredmény el nem tér 0-tól.

Amikor ez bekövetkezik, a program a következő utasítás végrehajtásával folytatódik.

Az utasításnak egy másik formája is létezik, amelyben az **y** nem szerepel. Ebben az esetben a program végrehajtása mindaddig szünetel, amig a megadott tárolórekesz tartalma és az **x** szám között létrehozott AND kapcsolat eredménye 0. Ha az eredmény 0-tól különböző, rátér a következő utasításra.

1.5. A számítógép felépítése

Nem kell félni, nem leszünk túlzottan műszakiak. Néhány dolgot azonban érdemes tudni a 64-es belső felépítéséről, ha a későbbiekben közölt rutinokat, trükköket meg akarjuk érteni.

Bizonyára sokan elgondolkoztak már azon, hogy miért reklámozzák a 64-est "64 k RAM"-mal, mikor a felhasználó csak 38 k-val rendelkezhet.

Nos, a 64 k valóban létezik, de nem használhatjuk mindet közvetlenül. A 6510-es mikroprocesszor összesen 64 k byte-ot képes megcímezni, ami azt jelenti, hogy 65536 különböző tárolóhelyet tud aktivizálni. A RAM ezek sze-

rint a számítógép összes lehetőségét kihasználná. Szükség van azonban még ROM-tartományra és a számítógép belső rutinjai által igénybevehető RAM területre is. Ilyen a már említett nulláslap, továbbá a BASIC-értelmező, az operációs rendszer és a karaktergenerátor ROM-jai. (Az utóbbiról a későbbiekben még szó lesz, előljáróban csak annyit, hogy a gép itt tárolja a képernyőjelek alakját.)

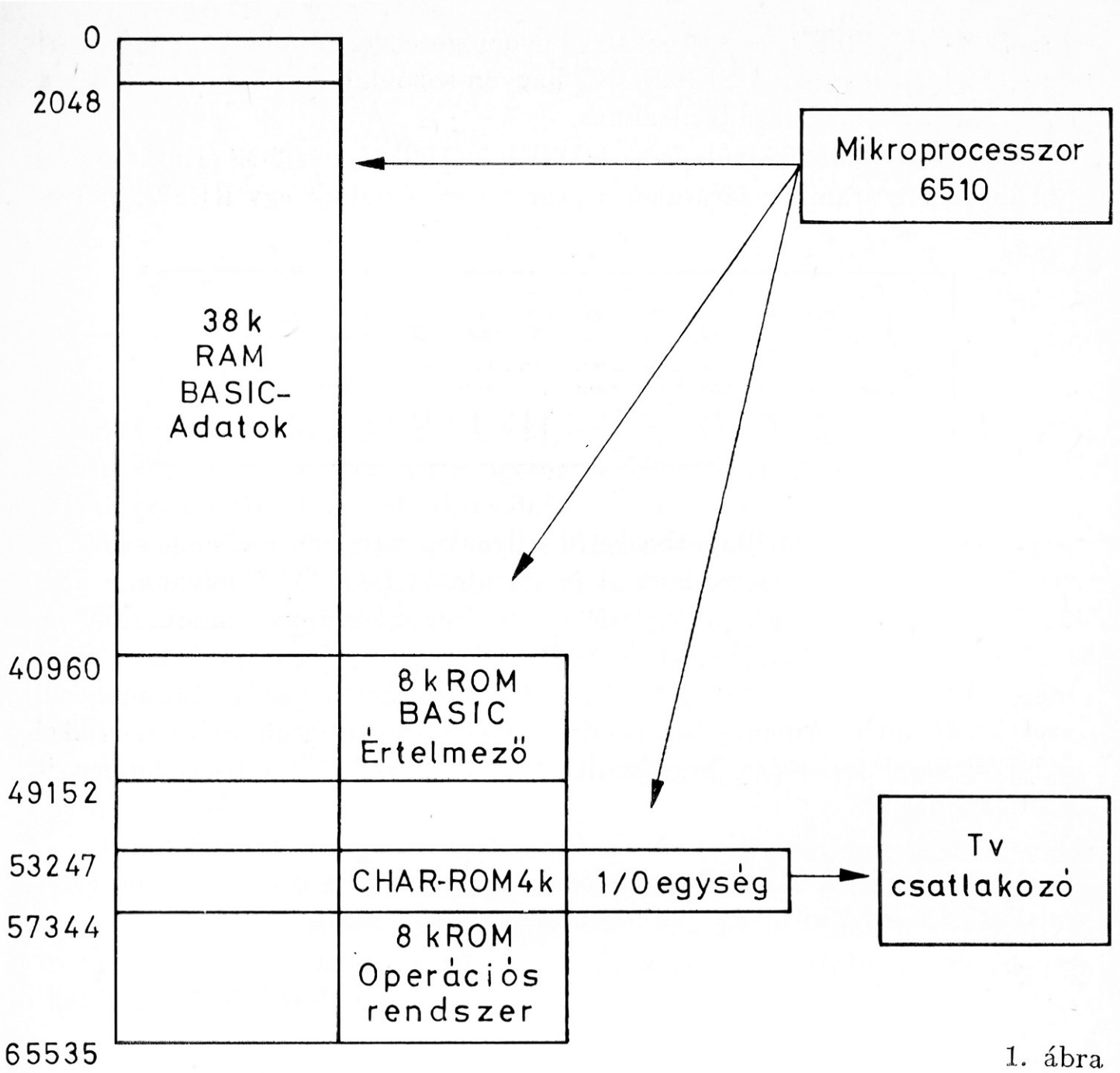
A ROM-tartomány terjedelme 20 k. A maradék 44 k-ból 2 k-t a video-RAM és a nulláslap foglal el, a 49152 címtől kezdődő 4 k RAM pedig a gépi kódú programoknak van fenntartva. Így a felhasználó számára valóban 38 k byte maradt.

Az 1. ábrán a számítógép egyszerűsített tárfelosztása látható. Eszerint a RAM és ROM egymás mellett helyezkedik el és ugyanazt a címtartományt foglalja le. Az, hogy éppen melyiket használhatjuk, az 1. tárolórekesz tartalmától függ. Ennek módosításával valósítható meg a ROM és RAM terület közti átkapcsolás. BASIC-ben sajnos erre nincs lehetőségünk, hiszen ha a BASIC–ROM-ot kikapcsolnánk, a processzor a programja helyett csupán üres tárolórekeszeket találna. Némi segítséget azért nyújt a CBM. BASIC-ből ugyanis a POKE és a LOAD utasításokkal bármikor írhatunk az egymást átfedő területekre, csupán ezek PEEK-kel és SAVE-vel való újbóli kiolvasása nem lehetséges. Ez azt jelenti, hogy az ilyen területre címzett POKE utasítás a PEEK-utasítás a ROM tartalmát olvassa ki.

A címtartomány felső harmadában marad még 4 k RAM. Ez a gépi kódú programnak van fenntartva, de adattárként a PEEK és POKE utasításokkal BASIC-ben is használható.

Végül van még az úgynevezett I/O (Input/Output) tartomány. Itt találhatók azok az építőelemek (IC-k), amik az interface-ekhez, a billentyűzethez, a képernyőhöz és a szintetizátorhoz tartoznak.

A processzor itt adja le azokat az információkat, amelyeket a megfelelő elemek pl. tv-jellé, hanggá vagy éppen a lemezegységre vonatkozó jellé dolgoznak fel és itt bonyolódik az ellenkező irányú adatátvitel is. Ide érkeznek a billentyűzetről vagy a perifériáról bevitt adatok. Ezenkívül itt található még a szín-RAM is. Amint az ábrán látható, itt a byte-ok háromszorosan is fedik egymást, mivel az I/O-elemek mindegyike saját RAM-területtel rendelkezik. Amikor pl. keretszín módosítás végett az 53280-as tárolórekeszt aktivizáljuk, akkor a byte nem a RAM-ba, hanem magába az IC regiszterébe kerül. Ez a szín-RAM-ra is vonatkozik. A 64-esnek összesen 4 darab I/O-eleme van. Ezek közül a VIC-et (grafika- és képernyővezérlés) valamint a SID-et (hangelőállítás) már ismerjük. Marad még a két CIA (Complex Interface Adapter) a billentyűzet és még néhány interface (pl. USER PORT) vezérlésére.



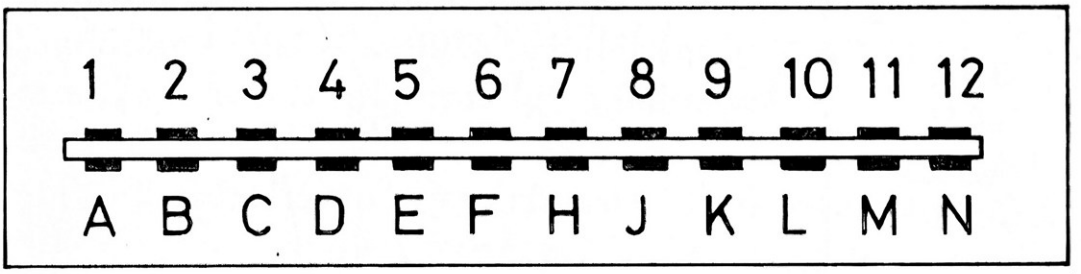
1. ábra

1.6. Saját kísérletekhez: RESET-gomb

Amikor a PEEK és POKE utasításokkal kísérletezünk, könnyen előfordulhat, hogy a számítógép valahol "kiakad". Ilyenkor általában a RUN/STOP-RESTORE billentyűkkel segíthetünk magunkon, de ha ez nem vezet eredményre, ki kell kapcsolnunk a gépet. Ilyenkor viszont elvesznek a segédprogramjaink. Azért, hogy ezt elkerüljük, érdemes készítenünk egy RESET-gombot. Ehhez szükségünk van egy USER PORT csatlakozóra (pl.: Cardcon

251–12–50–170 TRW) és egy egyszerű nyomógombra. Mindenképpen megéri a befektetést, hiszen a USER PORT nagyon sokoldalúan használható, így a csatlakozó egyéb célokra is alkalmas.

A nyomógombot kössük össze a USER PORT 1-es és 3-as érintkezőjével (2. ábra). Az áramkör zárásakor a processzor végrehajt egy RESET műve-



2. ábra

letet, azaz a gép alapállapotba kerül. Ilyenkor azonban csak az operációs rendszer számára szükséges byte-ok módosulnak. BASIC-ből ugyanaz a SYS 64738 utasítással érhető el. A tárolóban lévő gépi kódú programok, mint pl. a SIMON'S BASIC, továbbra is futtathatók, hiszen az áramellátás nem szűnt meg, tehát SYS xxxxx-szel újra indíthatók. Természeteen ehhez tudnunk kell ezek kezdőcímét! Amennyiben rendelkezünk olyan programmal, amelyikkel a NEW utasítás hatása "megfordítható", még a BASIC-programunkat is visszakaphatjuk.

Vigyázzunk! A RESET a mágneslemez meghajtót is inicializálja, ezért mielőtt erre sor kerülne, vegyük ki a benne levő lemezt.

2. A NULLÁSLAP

2.1. A nulláslap nem "nulla"

Aki már átnézte a CMB-kézikönyvet, annak bizonyára feltűnt a nulláslapra vonatkozó fejezet. Ez rengeteg trükköt és programozási lehetőséget kínál, csak ki kell tudnunk használni.

Az elnevezés – "nulláslap" – nem túl szerencsés. Szokás szerint a mikroproceszor első 256 byte-ját értik alatta, holott az első kilobyte-ról van szó. Célja és értelme gyorsan megmagyarázható. Az operációs rendszer és az értelmező a különböző állapotok, számok és kódok feljegyzéséhez regisztereket használ. Ahogy a diák az összeadásnál "fejben tartja" az 1-est ugyanúgy teszi ezt a számítógép a nulláslappal.

Az első 256 byte gyors adattárolásra alkalmas, mivel ezek címzéséhez egy byte is elegendő.

A nulláslap sok regiszterének meghatározott értéket kell tartalmaznia, hogy a gép rendeltetésszerű működését biztosítsa.

Mások egyáltalán nem használatosak, szabadon állnak a programozó rendelkezésére és vannak olyanok is, amelyeknek a tartalma célszerűen és hatásosan befolyásolható.

2.2. Mutatók (pointer) és vermek (stack)

Ezekkel a fogalmakkal sűrűn fogunk találkozni ezért némi magyarázatot igényelnek.

A mutatók (vagy vektorok) a tárolóterület meghatározott helyeire mutatnak, ahol információk vagy alprogramok állhatnak.

Például a kurzormutató a képernyőtárolóban a kurzor aktuális helyére és ezzel a villogó jel betűkódjára mutat.

Az alprogram mutatót azért vezették be, hogy lehetőséget teremtsenek az értelmező bővítésére. A jelek kivitelét végző rutin mutatóját módosítva

lehetőségünk nyílik pl. arra, hogy egy kis saját gépi kódú rutin alkalmazásával a PRINT utasítást úgy módosítsuk, hogy a jelek egyidejűleg a képernyőn is és a nyomtatón is megjelenjenek.

A mutatónak minden esetben meghatározott formátuma van. Általában 2 byte-ból, az ún. alsó (vagy LOW) és a felső (vagy HIGH) byte-ból áll.[1] Az aktuális kurzorpozíció megállapítására ill. a mutató által kijelölt tárolócím megállapítására a következő összefüggést használjuk:

cím=AB+256*FB

A számítógép sajátosságai közé tartozik, hogy a tárolóban az alsó byte mindig a felső byte előtt áll. Az egy byte-os mutatók egy meghatározott tárolótartományon belüli (pl. billentyűzet puffer, verem) aktuális pozícióra (0–255) mutatnak és értéküket a gép az ún. BÁZISCÍM-hez adja, ami az adot tartomány kezdőcíme.

A verem rendeltetése az adatok átmeneti tárolása és szükség esetén a bevitellel fordított sorrendű visszaadás.

Akárcsak egy valódi veremből, ebből is mindig csak a legfelső elem vehető ki és az új adatok is mindig a régiek "tetejére" kerülnek. A veremtároló a gép minden alprogramba való ugráskor használja. Itt rögzíti az elágazás helyét, amit az alprogram végén (RETURN) visszaolvas és így "megtudja" hol kell folytatnia a főprogram végrehajtását.

A 64-esnek három olyan veremtárolója is van, amelyek nem módosíthatók:

1. Processzor-verem gépi kódú programok részére (256–511).
2. Verem a BASIC alprogramok részére.
3. Verem a FOR ... NEXT ciklusok részére.

Ne csodálkozzunk azon, hogy az utóbbi kettőt nem találjuk a kézikönyv tárolótérképén, ezeket ugyanis különböző "hasznosított" területeken helyezték el.

Ez azonban senkit ne zavarjon.

Ennyi elmélet után végre rátérhetünk a gyakorlatra. A következő fejezetekben érdekes trükköket találunk (a szükséges elméleti kiegészítéssel együtt), amelyek a 64-es programozásnál segítségünkre lehetnek.

[1]Megjegyzés: A továbbiakban alsó byte= **AB**; felső byte= **FB**

Összefoglalás: mutató

cím = alsó byte+256 * felső byte
alsó byte = cím – INT (cím/256) * 256
felső byte = INT (cím/256)

A mutatók általában két byte-ból állnak, amelyek közül a tárolóban az alsó mindig megelőzi a felsőt.

3. A TÁR

3.1. A tár foglaltsági térképe

A 16. fejezetben megtalálhatjuk a CMB 64-es pontos tároló foglaltsági térképet (röviden: tártérkép). A nulláslap mellett az I/O tartomány listáját is közöljük. Felhívjuk a figyelmet arra, hogy a CMB-kézikönyvvel összehasonlítva néhány eltérést tapasztalhatunk. Pl. az állítólag szabad 673–767 tartomány első 5 byte-ja a CIA-által foglalt. Erre feltétlenül ügyelnünk kell, ha a nulláslapot adattárolóként akarjuk használni. Egy helytelenül alkalmazott POKE utasítással ugyanis könnyen tönkretehetjük az értelmezőt. Ennek ellenére érdemes kísérletezni. Sok olyan trükk van, amit véletlenül fedeztek fel, mások viszont célirányos kutatómunka eredményei. Általános vélemény szerint semmilyen POKE – kombináció nem vezethet a számítógép meghibásodásához.

Sikeres kísérletezést kívánunk!

3.2. A mágikus 1. byte

Ez a byte azért mágikus, mert mint már elmlítettük, a tárfelosztást vezérli. Erre a célra csak a 0 ...1 biteket használjuk. Alapesetben mindhárom értéke 1. Ha bármelyiket kikapcsoljuk, a tároló felosztása módosul. A 0. bittel a BASIC–ROM-ot (40960–49151), az 1. bittel pedig egyszerre a BASIC-et és az operációs rendszert kapcsolhatjuk ki. Ha mindkettő 0, akkor ezenkívül még az I/O területet is kikapcsoltuk, így összesen 62 kbyte RAM áll rendelkezésünkre (mivel a nulláslap és a VIDEORAM nincs átfedésben). A 2-es bit azt határozza meg, hogy a karaktergenerátor kiolvasható-e (háromszoros átfedés!).

Sajnos ennek a rendszernek van egy bökkenője. Ha ugyanis a BASIC-et és az operációs rendszert kikapcsoljuk, a gép kezelhetetlenné válik. Emiatt csak gépi nyelven férhetünk hozzá a rendkívül jól elrejtett RAM-hoz.

A karaktergenerátorral mindez kicsit másként alakul. Itt nincs olyan program, ami az operációs rendszer működéséhez szükséges lenne. Ennek

ellenére kezelhetetlenné válik a gép, ha a 2-es bitet kikapcsoljuk, hiszen nem férhetünk hozzá az I/O tartományhoz. Ugyanez vonatkozik a megszakító rutinra is, ami a billentyűzetet kérdezi le. Ilyenkor az a megoldás, hogy az 1.2 pontban ismertetett módon kikapcsoljuk a megszakítást.

Az alábbiakban bemutatunk egy gépi nyelvű programot, ami lehetővé teszi, hogy a tekintélyes 62 k-t legalább PEEK-ek és POKE-ok segítségével használhassuk. BASIC megfelelője is létezik ugyan, de ez *nem* működik. A könnyebb érthetőség kedvéért mégis közöljük mindkét listát:

```
1 REM *** P 2 ***
2 REM
10 REM !!! INDITANI SZIGORUAN TILOS !!!
15 REM ---------------------------------
20 POKE56334,PEEK(56334)AND254:REM MEGSZAKITAS KI
30 POKE 1,PEEK(1) AND 252:REM ROM KIKAPCSOLASA
40 POKE 2,PEEK(PEEK(251)+256*PEEK(252))
50 POKE 1,PEEK(1) OR 3:REM ROM BEKAPCSOLASA
60 POKE 56334,PEEK(56334) OR 1:REM MEGSZAKITAS BE
READY.
```

```
1 REM *** P 3 ***
2 REM
10 DATA 120,165,1,41,252,133,1,160,0,177
15 DATA 251,133,2,165,1,9,3,133,1,88,96
20 DATA 120,165,1,41,252,133,1,160,0,165
25 DATA 2,145,251,165,1,9,3,133,1,88,96
30 FOR I=680 TO 721:READ A:POKE I,A:NEXT
READY.
```

Nézzük meg közelebbről is a programot. A 20. és 60. sor már ismerős. A 30. sorban töröljük az 1. byte (1. regiszternek is nevezzük) 0-ás és 1-es bitjét. Ennek eredményeként használhatóvá válik a 62 k RAM. A 40. sorban kiolvassuk a kívánt rekesz tartalmát, amit átmenetileg a 2-es, egyébként nem használt rekeszbe írunk. Innen a ROM visszakapcsolása után is kiolvasható lesz. Ez sorrendben a következőképpen történik: A belső PEEK utasítások kiolvassák a 251-es és 252-es tárolórekesz tartalmát, amelyek a kiválasztott cím mutatójának alsó és felső byte-jait tárolják. Ezekből az előző fejezetben

ismertetett összefüggéssel létrejön maga a keresett cím, aminek tartalmát a külső PEEK utasítással kapjuk meg.

Tételezzük fel, hogy a szín-RAM alatt található 56000-es tárolórekeszt szeretnénk aktivizálni. A mutató felső byte-ját a következőképpen számítjuk ki:

```
FB=INT (56000/256)
```

Az alsó byte-ot úgy kapjuk, hogy a 16 bites 56000-ből kivonjuk a 8 bites FB-t. Tehát

```
AB=56000 - INT (56000/256)
```

A mutató beállítása a következőképpen történik:

```
POKE 251, AB:POKE 252, FB
```

SYS 680 után a keresett tárolótartalom PRINT PEEK (2) utasítással kiírható.

Az I/O-tartomány alatti RAM-ba, a többi átfedett területtől eltérően, normál POKE utasítással nem írhatunk. A ROM-ot ebben az esetben is ki kell kapcsolnunk. Az erre vonatkozó program majdnem azonos a P.2-vel, csak a 40. sort kell az alábbiak szerint módosítanunk:

```
40 POKE (PEEK (251) + 256 * PEEK (252)), PEEK (2)
```

Működése is gyakorlatilag ugyanaz, csak előzőleg a POKE-értéket rögzíteni kell a 2-es rekeszben. A mutatót az előzőek szerint állítjuk be. Ezt a programot a SYS 701 utasítással indítjuk.

A P.3-as program a P.2 gépi kódú megfelelőjének betöltőprogramja. (A módosított programra is vonatkozik.) A 10. sor tartalmazza a teljes PEEK – a 20-as a teljes POKE – programot. A két sor csupán 4 byte-ban különbözik. Ezek egymástól függetlenül használhatók és eltolhatók (ld. 40. sor). Ezt relokalizációnak (nem helyhezkötött) is nevezik.

Mindkét rutin hossza 21 byte.

Ezzel tehát 62 k RAM területet használhatunk, ebből 38 k-t BASIC programok és változók számára. A maradék 24 k területre a 4 k-s I/O tartomány kivételével (53248–57343) POKE utasításokkal írhatunk és a P.3 rutin segítségével a beírtakat újból kiolvashatjuk.

Mivel az előbbi rutin nem túl kényelmes, következzék most egy másik változata, amelyet a PRINT USR (cím) utasítással hívhatunk le. Az I/O terület alá POKE-kal történő beírásokhoz használhatjuk a következő utasításkombinációt:

SYS 715, cím, byte

Aki csodálkozik a rendhagyó írásmódon, annak elárulhatjuk, hogy a 64-es a gépi nyelven programozóknak ideális lehetőségeket nyújt. A ROM-rutinok kihasználásával saját utasításokat állíthatunk elő a leírt módon.

Ez a program is eltolható a tárban, de akkor a 70. sorban lévő USR-mutatót a megfelelő címre kell állítani. Ez ugyanúgy történik, mint a P.2 programban. A mutatónak mindig arra a címre kell mutatnia, ahol a 60. sorban levő FOR ... NEXT ciklus kezdődik.

```
1 REM *** P 4 ***
2 REM
10 DATA 165,20,72,165,21,72,32,247,183
15 DATA 1,20,165,1,41,252,133
20 DATA 1,160,0,177,20,168,165,1,9,3
25 DATA 133,1,88,104,133,21
30 DATA 104,133,20,76,162,179,32,253
35 DATA 174,32,138,173,32,247
40 DATA 183,32,253,174,32,158,183,165
45 DATA 1,41,252,133,1,138
50 DATA 160,0,145,20,165,1,9,3,133,1,96
60 FOR I=678 TO 747:READ A:POKE I,A:NEXT
70 POKE 785,166:POKE 786,2
READY.
```

Összefoglalás: Átfedések a tárolóban

Az egymást átfedő tártartományokat az 1. tárolórekesz (vagy regiszter) 0 ... 2 bitjei vezérlik. Ezek értéke alapállapotban 1. Törlésükkel az alábbiak szerint módosíthatjuk a tárfelosztást:

0-ás bit törlése	– a BASIC–ROM kikapcsolása
1-es bit törlése	– a BASIC és az operációs rendszer kikapcsolása

0-ás és 1-es bit törlése – a BASIC, az operációs rendszer és az I/O kikapcsolása

2-es bit törlése – lehetővé teszi a karaktergenerátor kikapcsolását (BASIC-ből a megszakító rutin kikapcsolása után lehetséges)

3.3. A tár védelme

Miután feltártuk a 62 k-s tároló elérési lehetőségeit, tegyük most ennek éppen az ellenkezőjét: szűkítsük le a BASIC-tárolót úgy, hogy bizonyos területeihez és az itt tárolt adatokhoz az értelmező ne férjen hozzá. Rögtön felmerülhet a kérdés, hogy ez mire jó? Képzeljük el, hogy egy 8 sprite-tal dolgozó programot akarunk írni. Négynek a definícióját elhelyezhetjük a 11-es, 13-as, 14-es és 15-ös blokkban. A 15-ös blokk után azonban kezdődik a VIDEORAM és a BASIC-tároló, amelyeket nem ajánlatos felülírni. Mi következik ebből? Át kell helyeznünk a BASIC-tárolót, hogy a sprite-ok számára helyet teremtsünk. A 43-as és 44-es tárolórekeszben (nulláslap) található a BASIC-tároló kezdőcímének mutatója (AB és FB).

Ha a BASIC-programunkat pl a 2560-as tárolócímtől kezdődően akarjuk elhelyezni, a 43/44-es rekesz tartalmát az ismert módon meg kell változtatni:

```
POKE 43, 2561 - (INT 2561/256) * 256
POKE 44, (2560+1)/256
```

1-et azért kell hozzáadnunk, hogy a mutató az első programsor elejére álljon. A BASIC-program első bytejának mindig 0-nak kell lennie, ezért:

```
POKE 2560,0
```

Hátra van még a CLR, hogy a változók, tömbök és hasonlók mutatóit is hozzáigazítsuk az új helyzethez, mert ezek még a régi BASIC kezdőcímre (2048) mutatnak.

A BASIC–RAM végének leszállításához hasonlóképpen járunk el. Ebben az esetben az 55/56-os regisztereket használjuk és ezzel megtakaríthatunk egy POKE cím 0, utasítást.

Sajnos ennek az eljárásnak egy nagy hátránya van. A mutatók átállításával ugyanis a BASIC tárolóban lévő programszöveg nem tárolódik el automa-

tikusan az új tárolótartományba, ezért ezeket az utasításokat még a program begépelése ill. betöltése előtt kell bevinnünk. A legegyszerűbb módszer egy kis betöltő program készítése, amely átállítja a mutatókat és azt követően behívja a főprogramot. Az alábbi példa ezt szemlélteti:

```
1 REM *** P 5 ***
2 REM
10 POKE 43,2561-INT(2561/256)*256
20 POKE 44,(2560+1)/256:POKE 2560,0:CLR
30 LOAD"FOPROGRAM",8
READY.
```

Amikor ez a program fut, a 64-es tulajdonképpen valami lehetetlent csinál. A két első sor felemeli a BASIC-tároló kezdőcímét. Ennek az az eredménye, hogy a program a továbbiakban nem listázható ki, tulajdonképpen az értelmező számára már nem is létezik. Ennek ellenére a többi sort is feldolgozza, kivéve a kisebb sorszámra történő ugrásokat és hasonlókat, mivel ilyenkor minden sort az aktuális mutatóhelyzet után keres így az előző sorokat nem találja meg. A LIST-utasítás a "FŐPROGRAM" listáját eredményezi.

A LOAD-utasításnak is van egy kis trükkje. Amikor egy programon belül kerül feldolgozásra, akkor a betöltés után a számítógép nem a régi programot folytatja, hanem elkezdi az új program végrehajtását. A főprogram tehát automatikusan elindul. A mutatókat mindig, még a program begépelése előtt, "kézzel" kell átállítanunk, nehogy a sprite-ok vagy egyebek felülírják a keservesen bevitt programsorokat. Ha kész a program és már csak a mutatók beállítása van hátra, rögzítsük mágneslemezre vagy kazettára, ahonnan a P.5-ös betöltőprogrammal újból beolvashatjuk.

A következő fejezetben még bemutatunk néhány alkalmazási példát, amelyeknél a BASIC-tároló egyes részeit védenünk kell.

Összefoglalás: Tároló védelme

A BASIC-tár kezdőcímének felemelése.

POKE 43, AB:POKE 44, FB:POKE cím, 0:CLR

A BASIC-tár végének leszállítása.

POKE 55, AB:POKE 56, FB:CLR

3.4. Szabad tár

Erről már sokat hallottunk, de a biztonság kedvéért néhány szó erejéig mégegyszer térjünk rá: FRE(0)-függvény problematikájáról van szó.

Ha a szabad tárkapacitás kisebb, mint 32 768 byte, akkor a PRINT FRE(0) utasítás eredménye egy pozitív szám, ami megegyezik a szabad byte-ok számával. Ellenkező esetben viszont (pl. bekapcsoláskor) egy negatív érték, ami látszólag semmiféle összefüggésben nincs a tárkapacitással.

A FRE-függvény eredménye mindig egész szám (integer). A BASIC egész típusú változóinak értéktartománya: – 32767 –+ 32768. Ennél nagyobb számot (pl. 68000) az értelmező kénytelen negatív számmal kifejezni. A szabad byte-ok számát ebben az esetben az alábbi összefüggésből kapjuk meg:

PRINT 65536+FRE (0), ha FRE (0),< 0

Most más téma. Gyakran előfordul, hogy néhány adatot egy gépi program számára tárolni akarunk, vagy nem akarunk egy egész változót "elpocsékolni" egy bytera vagy bitre. Az is elképzelhető, hogy a program változóit törölni szeretnénk (CLR), de néhány vezérlőváltozó tartalmát feltétlenül meg kell tartanunk. Mit csináljunk?

Ilyenkor egyetlen lehetőségünk van. Keresünk egy szabad tárolóterületet a nulláslapon és POKE-kal beírjuk ide az adatokat. Erre a tárolótartományra sem a CLR, sem a NEW nincs hatással. Az alábbiakban felsoroljuk a nulláslap szabad területeit:

Szabad byte-ok

2	
251–254	adott esetben módosítható
678–767	
780–827	csak ha SYS-t nem alkalmazunk
820–1019	szalagegység használatakor felülíródik
1020–1023	
2024–2039	

4. ADATHALMAZ TÁROLÁSA ÉS A PERIFÉRIÁK

4.1. Grafika, képernyő és egyebek tárolása

A BASIC – értelmező SAVE – rutinja nem a legkényelmesebb, grafika vagy gépi program tárolásakor pedig végképp csődöt mond. Ilyenkor ugyanis meg kell adnunk a rögzítendő tárolótartomány kezdő- és végcímét is. Szerencsére mint mindig, ebben az esetben is van egy trükk – igaz, egyelőre csak kazettás egységre –, amivel az ún. "mesterséges" file-okat (=szalagra vagy mágneslemezre rögzített feljegyzések) tárolhatjuk.

Mint már sokszor, most is a nullászlap lesz segítségünkre. A 170–195-ig terjedő tárolótartományban találhatók az adatfeldolgozással kapcsolatos mutatók és regiszterek.

Ezek közül legfontosabbak a lerögzítendő tartomány kezdő- és végcímének mutatói. A kezdőcím mutatója a 193/194-es, a végcímé a 174/175-ös regiszterben található. A SAVE – utasítás SYS 62954-gyel hívható. Kell még adnunk egy nevet, amit legcélszerűbben a program elejére, egy REM-sorba írhatunk. Hogy a file-név hol áll, azt az operációs rendszer a 187/188-as regiszterben lévő mutató alapján állapítja meg.

Végül szükségünk van még egy másodlagos címre, a külső tárolóegység készülékszámára és a file-név hosszára. Legjobb, ha megnézzük a következő programot:

```
1 REM *** P 6 ***
2 REM
10 REM FILE-NEV
20 POKE 193,SA:POKE 194,SF:REM KEZDOCIM (AB/FB)
30 POKE 174,VA:POKE 175,VF:REM VEGCIM   (AB/FB)
40 POKE 187,PEEK(43)+6:POKE 188,PEEK(44):
   REM FILE-NEV MUTATO
50 POKE 183,H:REM A FILE-NEV HOSSZA
60 POKE 186,1:POKE 185,0:REM KESZULEKSZ.
   /SZEKUNDERCIM
70 SYS 62954:REM SAVE RUTIN A ROM-BOL
READY.
```

Az így rögzített programfile-ok, grafikák és hasonlók a LOAD "file-név", 1,1 utasítással ugyanarra a tárterületre tölthetők vissza, ahonnan behívtuk, hiszen a kezdőcímet is rögzítettük. Néhány szó a 40. sorról: Ez a sor közli az operációs rendszerrel a file-név helyét. Mivel ez az első sorban a REM után áll, ezért a BASIC – program kezdőcíméhez 6-ot kell adnunk, hogy a pozícióját meghatározzuk. (A lista 1. és 2. sora nem tartozik a programhoz, csak a könyvben való tájékozódást szolgálja! – a magyar vált. készítői.)

Amennyiben a PEEK (43) + 6 kifejezés értéke 255-nél nagyobbra adódna,

ILLEGAL QUANTITY ERROR (= nem megengedett mennyiség)

hibaüzenetet kapunk. Ebben az esetben kiszámítjuk a címet is

PEEK (43) + 6 + PEEK (44) * 256

összefüggés szerint és a már ismert módon visszaszámoljuk belőle AB és FB értékét, amelyeket a megfelelő regiszterekbe írunk.

Mágneslemez meghajtó esetén minden sokkal egyszerűbb. Létezik egy értelmes BASIC-függvény, amellyel majdnem olyan jól írhatunk a lemezre, mint POKE-kal a tárolóba. A PRINT#1, CHR$(x) – kombinációról van szó. Ez pontosan egy ASCII – jelet küld a lemezegységnek. Ahhoz, hogy értelmesen tudjuk használni, ismernünk kell a programfile-ok rögzítésének formáját. Minden program egy tartalomjegyzék – bejegyzésből és a programszövegből áll. A szöveg elején álló két byte adja meg a töltőfolyamat kezdőcímét. Ez normál esetben 1 és 8 (1+256*8=2049 – a BASIC tároló kezdőcíme). A programszöveg byte-onként, értelmezőkódok sorozataként, rögzített. A lemezegység minden byte-ot, rendeltetésétől függetlenül, ASCII-kódként kezel. Ha tehát kiolvasunk egy mutatót, amelynek két byte-ja 65 és 66, akkor a GET#1, A$, B$ utasítás végrehajtása után az A$-változó tartalma "A", B$-é pedig "B" lesz. (Az A-betű ASCII-kódja 65, a B-betűé 66.) A két füzér ASCII-kódjának lekérdezésével megkapjuk a két byte-ot.

Ha a PRINT#1, CHR$ (x) utasítással egy jelet küldünk a lemezegységnek, akkor az x-szám a lemez aktuális helyén rögzírésre kerül. Ezzel szemben a PRINT#1, X utasítás az X-et több byte hosszú sorként (számjegyenként 1 byte) rögzíti.

Ebből adódik a programfile-ok előállításának egy egyszerű módszere:

1. Tartalomjegyzék (directory) – bejegyzés elvégzése:

Az OPEN – utasítás végzi.

2. Kezdőcím beírása POKE-kal a következők szerint:

PRINT#1, CHR$ (AB):PRINT#1, CHR$ (FB)

3. Szöveg tárolása:

A szöveg lehet például képernyőgrafika is. A rögzítés byte-onként történik.

A következő programmal lemezre rögzíthetjük a képernyő tartalmát:

```
1 REM *** P 7 ***
2 REM
10 OPEN 1,8,1,"0:KEPERNYO"
20 PRINT#1,CHR$(0);
25 PRINT#1,CHR$(4);:REM KEZDOCIM
30 FOR I=1024 TO 2023
40 PRINT#1, CHR$(PEEK(I));
50 NEXT I:CLOSE 1
READY.
```

Az eredményt most is LOAD "KÉPERNYŐ",8,1 utasítással tölthetjük vissza.

A 10. sorban csak a file-név ("0:" után) módosítható. Nagyon fontos az 1-es mésodlagos cím, ami a DOS-sal (disk operációs rendszer) közli, hogy tárolni akarunk. Sose felejtsük el a program végéről a CLOSE 1 utasítást, nehogy tönkretegyük a file-unkat!

A file-t úgy írhatjuk felül, hogy az idézőjelben a "0" elé egy "@"-jelet teszünk.

4.2. MERGE, vagyis a programok összefűzése – "kézzel"

Térjünk rá egy gyakran előforduló problémára. Mi van akkor, ha a rendelkezésünkre álló, kipróbált rutinjainkból egy csodálatos programot akarunk összeállítani? Ha nem rendelkezünk egy MERGE-segédprogrammal, ami összefűzi a programrészeket, úgy tűnik, nem marad más hátra, mint újból mindent begépelni. Most bemutatunk egy eljárást, amivel a részprogramok segédprogram hiányában "kézzel" összefűzhetők. Csupán néhány egyszerű utasítás kell hozzá.

Ha a problémát közelebbről megvizsgáljuk, kiderül, hogy az egésznek az a lényege, hogy a másodikként bevitt program az elsőt felülírja. Ahhoz, hogy ez ne következzék be jó lenne, ha közölhetnénk az értelmezővel, hogy a második részt hová töltse be. Ez nem olyan egyszerű, ezért logikusan következik, hogy a régi program által igénybevett tártartományt az értelmező számára hozzáférhetetlenné kell tennünk – ezt viszont már ismerjük!

Ha egy tárterületet védelem alá helyeztünk, az új programrészt egy sima LOAD-dal betölthetjük. Ezután egyszerűen megszüntetjük a védelmet. Fontos, hogy a másodikként betöltött programrész kezdő sorszáma az első programrész utolsó sorszámánál nagyobb legyen. Ellenkező esetben kilistázható ugyan az összefűzött program, de az értelmező mégsem tudja végrehajtani.

Nézzük az eljárást:

1. PRINT PEEK (43), PEEK (44)

Ezzel megkapjuk a BASIC-program kezdőcímének mutatóját. (Alapesetben 1 és 8) Ezt a két számot (AB és FB) jegyezzük fel, mert később vissza kell majd állítanunk az eredeti állapotot.

2. POKE (43), (PEEK (45) + 256 * PEEK (46)–2) AND 255

POKE (44), (PEEK (45) + 256 * PEEK (46)–2/256

A 45-ös és 46-os tárrekeszben található a változótartomány kezdőcíme. 2 byte-tal előtte fejeződik be a BASIC-program.

A változótartomány mindig pontosan a programszöveg mögött található, mivel a programsorok módosításakor megfelelően el kell tolódnia. A két

POKE – utasítással beírtuk a 43/44-es rekeszbe az új BASIC-kezdőcímet (legalábbis az értelmező számára), ami az első BASIC-program végcímét közvetlenül követő byte. Ezáltal az előtte levő tartomány védetté vált, az új program nem írja felül.

3. NEW

Mivel a többi mutatót hozzá kellett igazítanunk az új helyzethez, valamint a megmaradt változókat az értelmező nemkívánatos programsorokként értelmezheti, a megmaradt tártartományt NEW-val inicializálni kell. Az előző programot ez, a védelem miatt, nem érinti.

4. LOAD

Most már egy egyszerű LOAD utasítással beolvashatjuk a következő részprogramot. Ebben az esetben semmiképpen sem használhatjuk a

LOAD "név",X,1

utasítássort, hiszen akkor a gép, figyelmen kívül hagyva a mutatókat, mégis felülírná a régi programot. Ilyenkor a lemez tartalomjegyzékét is betölthetjük anélkül, hogy a tárban levő program elveszne, ezt azonban a következő lépés előtt NEW-val törölni kell, mert bizonyára nem akarjuk hozzáfűzni az előző programhoz.

Ha az egész folyamatot végigcsináltuk, vissza kell állítani az eredeti állapotot.

5. POKE 43, AB:POKE 44, FB

ahol AB és FB a 43/44-es rekesz eredeti tartalma (1. pont).

Ezzel ismét visszaáll az eredeti állapot. A két program ezután már együtt szerepel, együtt futtatható és tárolható.

Összefoglalás: MERGE – "kézzel"

1. PRINT PEEK (43), PEEK (44) – számokat megjegyezni (AB és FB)!
2. POKE 43, (PEEK (45) + 256 * PEEK (46)–2) AND 255
   POKE 44, (PEEK (45) + 256 * PEEK (46)–2)/256
3. NEW

4. LOAD "név", X (,ahol X az aktuális készülékszám)

5. POKE 43, AB:POKE 44, FB

A műveleteket pontosan a leírt módon és sorrendben kell elvégezni!

4.3. A tartalomjegyzék (directory)

Az ebben a fejezetben leírtak sajnos kizárólag a VC–1541 típusú mágneslemez meghajtóra érvényesek. Akinek nincs ilyen készüléke, nyugodtan lapozza át ezt a részt.

Az első trükkel az előző fejezetben már megismerkedtünk: a tartalomjegyzék betöltése a tárban lévő program elvesztése nélkül. Használhatjuk úgy is, hogy programmal betöltjük és pl. tömbként tároljuk (pl. adatfeldolgozó programokban ...). Ilyen program található a VC–1541-es kézikönyvben (Függelék) és a TEST/DEMO lemezen, "DIR" néven. Könnyen átírhatjuk saját céljainkra. Érdekes a tartalomjegyzék szerkezete. A kézikönyv erről is megfelelő információkat nyújt. Figyelemre méltó magának a betöltésnek a folyamata:

OPEN 1,8,0, "$0"

(óvatosságból egy kísérleti lemezt használjunk!) utána

GET#1, A$, B$

a tartalomjegyzéket byte-onként olvassa be. A kézikönyv ellen szól, hogy néhány a lemezegységre vonatkozó utasításformát meg sem említ. Így pl. azt, hogy a tartalomjegyzéket bizonyos szempontok szerint is betölthetjük.

Ezek közül legegyszerűbb a

LOAD "$$",8

ami csak a lemez nevét és a szabad blokkok számát tölti be.

Megtehetjük, hogy csak meghatározott bejegyzéseket (pl. ABC-vel kezdődő file-okat) keresünk a tartalomjegyzékben. Ehhez a

LOAD "$:ABC*",8 utasítássort használhatjuk.

Ehhez hasonlóan járunk el, ha a keresést a file típusa szerint végezzük. Az utasítássor ekkor a következőképpen alakul:

LOAD "$:* = a file típusa",8

ahol a file típusát kezdőbetűjével jelöljük (Prg, Seq, Rel, Usr). A gép ilyenkor is beolvassa a teljes tartalomjegyzéket, de csak a megfelelő típusú, vagy egyéb szempontnak megfelelő file-okat listázza ki. Ugyanígy működik pl. a file-okat törlő (SCRATCH) utasítás is. Pl. a

PRINT#15, "S:* = S"

minden szekvenciális file-t töröl.

Végezetül még egy trükk, amivel pénzt takaríthatunk meg. A lemezmeghajtók normál esetben egyoldalas (single sided) lemezzel dolgoznak. A legtöbb ilyen lemeznek azonban, egy kis átalakítással, mindkét oldala használható. Ehhez még egy írásvédelmet nyújtó bevágást kell készítenünk a lemez másik szélén, az eredetivel szemben. Ez legkönnyebben egy hagyományos lyukasztókészülékkel oldható meg, úgy, hogy átjelölés után egy fél lyukat vágunk a lemez szélébe (3. ábra). Az így nyert új tárolóterület használhatóságát a TEST/DEMO lemezen lévő CHECK–DISK programmal ellenőrizhetjük. Ez a formált lemez összes sávját teleírja adatokkal, majd ezt az olvasási hibák szempontjából ellenőrzi. Az esetleg hibás blokkot jelzi a képernyőn. A program lefutása után (ami sajnos nagyon nagyon hosszadalmas), a tartalomjegyzék még mindig üres, de megjelenik a

0 BLOKS FREE

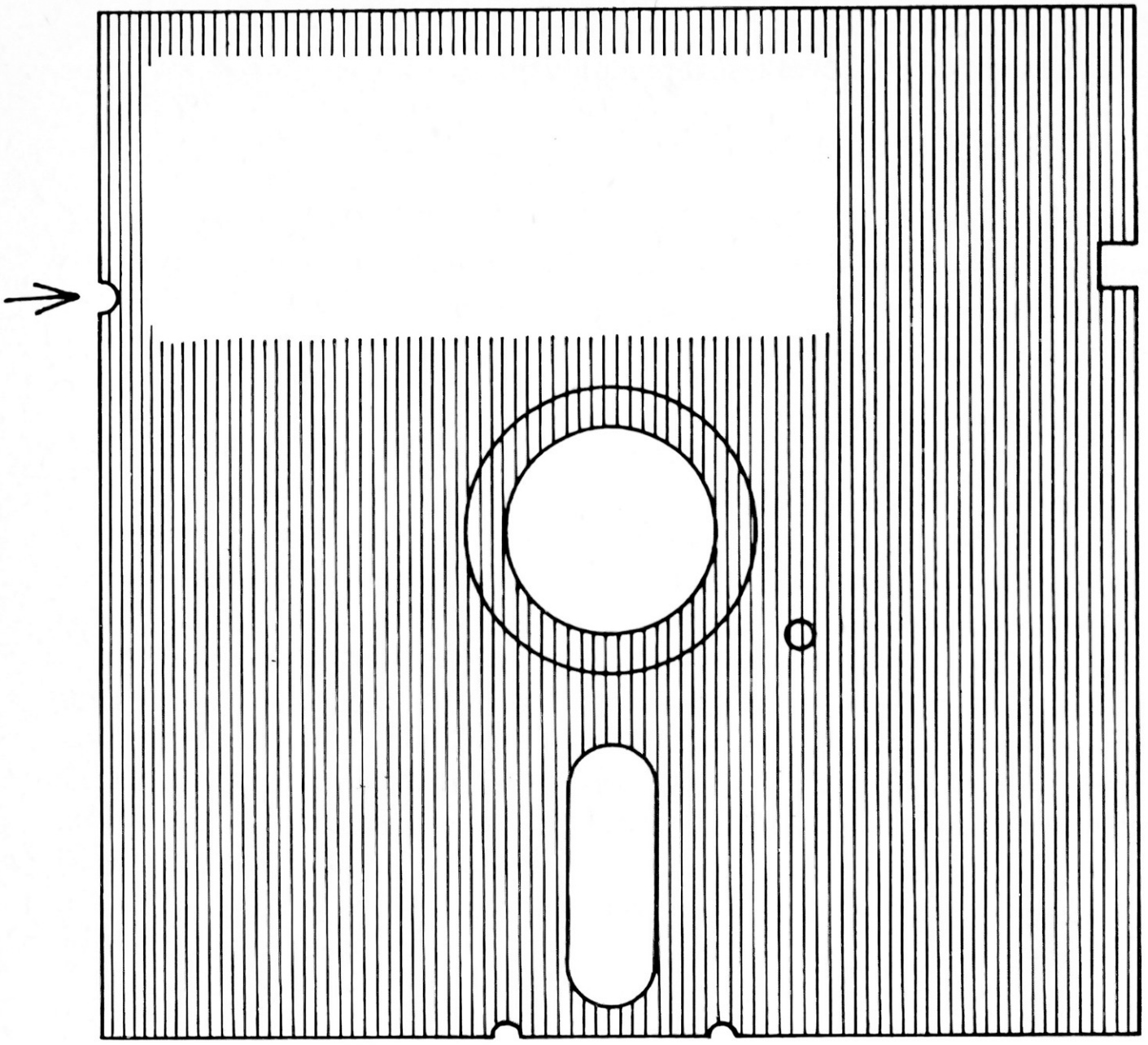
kijelzés, ami az

OPEN 1,8,15, "V": CLOSE 1

utasításokkal módosítható. Ezután ismét mind a 664 blokk rendelkezésünkre áll.

Összefoglalás: A tartalomjegyzék

LOAD "$$",8	– csak a lemez nevét és a szabad blokkok számát tölti be
LOAD "$:ABC*",8	– csak azokat a file-okat tölti be, amelyeknek a neve ABC-vel kezdődik.
LOAD "$:* = típus",8	– csak az adott típusú file-okat tölti be



3. ábra

4.4. Vegyes apróságok a perifériákkal kapcsolatban

A "nagyobb" trükkök után következzék most néhány kisebb PEEK és POKE utasítás, amelyek az adatok be- és kivitelét segítik.

Néha hasznos lehet, ha tudjuk a megnyitott file-ok számát.

Mint tudjuk, egyidejűleg max. 10 lehet nyitva. Ha a 11.-et is megkíséreljük megnyitni, a számítógép

TOO MANY FILES OPEN ERROR (= túl sok nyitott file)

hibaüzenettel reagál. A megnyitott file-ok számát a 152-es rekesz tárolja, amit előzőleg kiolvasva (PEEK (152)) elkerülhetjük a hibát.

A CMD-utasítással a képernyőről a perifériára irányíthatjuk a kivitelt. A készülék fizikai egységszámát (1=kazetta, 4=nyomtató, 8=lemezegység) a 154-es rekesz őrzi. Ennek tartalma normál esetben 3, tehát a CMD-utasítást a POKE 154,3 utasítással a file megsértése nélkül semlegesíthetjük. A kivitel a POKE 154, X utasítással ismét a perifériára irányítható.

Hasonlóan működik a 153-as rekesz is. Ez az aktuális beviteli készülék számát tárolja. Ha pl. a számítógépet egy V.24-es interface-en keresztül akarnánk távvezérelni, itt egy 2-es állna. Perifériáról való adatátvitel esetén értéke mindig a megfelelő készülékszám, normál esetben (billentyűzet) pedig 0.

Érdekes a 184-es rekesz is. Itt található a legutolsóként használt file száma. Másodlagos címét a 185-ös rekesz tárolja. Ebből megállapítható, hogy pl. a nyomtatót egy meghatározott üzemmódban használták-e, és hasonlók.

A csoport harmadik tagja a 186-os tárrekesz, amiben a legutolsóként használt készülék fizikai egységszáma van.

Ezt a címet olyan programokban használhatjuk, amelyeknek a számítógép pillanatnyi felszereltségétől függően kell vagy a szalagegységhez vagy a lemezegységhez férnie. Betöltés után a program a 186-os rekesz tartalmából megállapítja, hogy melyik perifériát használtuk és ezután ennek megfelelően dolgozhat tovább.

A 147-es rekesz tartalma 0 vagy 1 lehet. Azt rögzíti, hogy a lemezegység vagy a kazettaegység számára utolsóként kiadott olvasási utasítás LOAD (=0) vagy VERIFY (=1) volt-e. A még nyitott file-ok a SYS 62255 utasítással ismét lezárhatók.

Az utolsó trükk csak a kazettásegységet (Datasette) használóknak szól. A 150-es tárrekesz a kazettamotor állapotjelzőjét tárolja. Ha a motor nincs bekapcsolva, értéke 0, egyébként 0-tól eltérő.

Összefoglalás: Perifériatrükkök

PRINT PEEK (152) : a nyitott file-ok száma
PRINT PEEK (153) : az aktuális beviteli készülék száma
PRINT PEEK (154) : az aktuális kiviteli készülék száma
PRINT PEEK (184) : az aktuális file
PRINT PEEK (185) : az aktuális file másodlagos
PRINT PEEK (186) : az aktuális készülékszám
PRINT PEEK (147) : az utolsó olvasási utasítás
SYS 62255 : minden file lezárása
PRINT PEEK (150) : a kazettamotor állapotjelzője

4.5. Az ST állapotváltozó

Bizonyára mindenki hallott már az ST állapotváltozóról. Ez jegyzi, ha kazetta – üzemmódban a LOAD és VERIFY utasítások esetében, illetve a soros busz használatakor hiba merül fel. A hiba jellegétől függően a változó más – más értéket vesz fel, azaz mindig más állapotbit értéke lesz 1. Amennyiben semmiféle hiba nem fordult elő, ST=0. A soros busszal kapcsolatos hibák felmerülésekor ST=–128, ami a

DEVICE NOT PRESENT ERROR

hibajelzést eredményezi. Íráshibánál ST=1, olvasási hibánál ST=2. Kazetta és mágneslemez esetén is ST=64, ha a gép az adatfile végére ért.

A szalag végét (kazettánál) – 128-cal, az ellenőrzési hibát 32-vel jelzi. Ez a hiba akkor is felmerülhet, ha a műveleteket rendesen és hibaüzenet nélkül végrehajtotta. Ha valamelyik hibát nem tudta kijavítani (LOAD és VERIFY esetén az ellenőrzőblokk által), akkor a 4. bitet (ST=16) kapcsolja be. A blokk hosszával kapcsolatos hibák a 2. (ST=4) és a 3. (ST=8) bitet aktivizálják. Amikor egyszerre több hiba is előfordul, akkor a megfelelő bitek bekapcsolódnak és értékük összegződik. Így ha ellenőrzési hiba van és a szalag végére is értünk ST=–128+32=–96.

Az állapotbyte (ST) értékének kiolvasásával tehát a hiba típusa mind szalagegység, mind lemezegység esetén könnyen megállapítható.

5. A KÉPERNYŐ

Nem kell mindig nagyfelbontású grafika ahhoz, hogy egy jó képet programozzunk, éppen ezért ebben a fejezetben a normál képernyőépítéssel és annak kezelésével foglalkozunk.

5.1. Negyedpont grafika

Néztük-e már közelebbről a 64-es grafikus jeleit? Vannak köztük olyanok, amelyek pontosan negyed- ill. feleannyi helyet foglalna el, mint egy normál képernyőjel. Vegyük hozzá még a reverz jelkészletet és máris vannak háromnegyed és egész nagyságú jeleink is. Mivel a képernyő 25 sorból áll és minden sorba 40 jel fér, ezzel a negyedpont grafikával 50×80 pontot használhatnánk. Miután ezek a grafikus jelek a Commodore (C=) – billentyűvel aktivizálhatók, kisbetűs üzemmódban is használhatók.

Az lenne a legjobb, ha egy alprogrammal lehívhatnánk őket. Igen ám, de ha már elhelyeztünk egy pontot (ill. esetünkben valamilyen grafikus jelet), akkor ugyanabba a tárrekeszbe POKE-kal nem írhatunk egy más, negyedpontnyi jelet is, hiszen ezzel felülírnánk a régit. Valahogy be kellene építeni a régi jelet az újba.

Az egyik lehetőség az, hogy a teljes képernyőtartalmat egy tömbbe másoljuk, majd a pontokat egyenként hozzátesszük. Ezután egy alprogrammal a tömb tartalmát ismét képernyőkódokká alakítjuk.

Ha figyelembe vesszük, hogy a negyedpontnyi grafikus jelek elhelyezkedésének minden lehetséges kombinációja létező jel a 64-es karakterkészletében, egy egyszerűbb megoldás is adódik.

A képernyőjelek és pontok összes lehetséges kombinációja számára táblázatba foglalhatjuk a POKE-kal beírandó byte-okat. Ezt a táblázatot betöltjük a BASIC-tárolóba, így a gép a blokkgrafika-rutin minden lehívásakor kikeresheti a táblázatból a kívánt jelet. Az alábbiakban ismertetünk egy programot, amivel ilyen negyedpont grafikát állíthatunk elő.

Az első részben létrejön az inicializálás, azaz a szükséges táblázat beolvasása. Ez egy kissé terjedelmesre sikerült, de egy törlő alprogramot is tartalmaz.

A 60000. sortól kezdődik a jelek elhelyezését és törlését végző alprogram.

Az előzőbe GOSUB 60000-rel, az utóbbiba GOSUB 60001-gyel ugrik.

Ha az L-változó értéke 0 (60000. sor), a táblázat első fele (= rajzolás), ha 1 (60001. sor), a második fele (= törlés) kerül felhasználásra.

A kijelölt pont koordinátáit az X(0–49) és Y(0–79), a színkódot pedig az SZ változóban kell megadni.

A program listája:

```
1 REM *** P 8 ***
2 REM
3 PRINT "■":SZ=14
10 DIM PD%(1,1,1,15),P2%(15)
20 FOR B=0 TO 1:FOR X=0 TO 1:FOR Y=0 TO 1:
   FOR Z=0 TO 15
30 READ PD%(B,X,Y,Z):NEXTZ,Y,X,B
40 FOR I=0 TO 15:READ P2%(I):NEXTI
50 DATA 126,126,226,97,127,97,251,226,252,127,236,
   160,252,251,236,160
51 DATA 123,97,255,123,98,97,254,236,98,252,255,
   254,252,160,236,160
52 DATA 124,226,124,255,225,236,225,226,254,251,
   255,254,160,251,236,160
53 DATA 108,127,225,98,108,252,225,251,98,127,254,
   254,252,251,160,160,
54 DATA 32,32,124,123,108,123,225,124,98,108,255,
   254,98,225,255,254
55 DATA 32,126,124,32,108,123,225,226,108,127,124,
   225,127,251,226,251
56 DATA 32,126,32,123,108,97,108,126,98,127,123,
   98,252,127,97,252
57 DATA 32,126,124,123,32,97,124,226,123,126,255,
   255,97,226,236,236
58 DATA 32,126,124,123,108,97,225,226,98,127,255,
   254,252,251,236,160
60000 L=0:GOTO 60010
60001 L=1
60010 Y=49-Y:O=INT(X/2):S=INT(Y/2):PO=O+40*S
60020 X1=X-2*O:X2=Y-2*S:X3=PEEK(1024+PO)
60030 F=0:FOR I=0 TO 15: IF X3=P2%(I) THEN X3=I:F=1
```

```
60040 NEXT I
60050 IF F=0 THEN X3=0: IF L=1 THEN RETURN
60060 POKE 1024+PO,PD%(L,X1,X2,X3):POKE 55296+PO,
      SZ :RETURN
READY.
```

A 60010-es sor kiszámítja az általunk megadott pont képernyőtárolón belüli helyzetét, a 60020-as kiolvassa az itt található (régi) jel képernyőkódját. Ezt betölti az X3-változóba. Ezenkívül kiszámítja az általunk megadott pont jelraszteren belüli helyét (bal/jobb = X1; alul/felül = x2). A 60030-as sor a régi jel képernyőkódja és az utolsó DATA-sor alapján megállapítja, hogy az új jel a táblázat melyik oszlopából származzon, majd az oszlopszámot az X3-változóba tölti.

Ha a régi jel nem grafikus jel (kódja nem szerepel az utolsó DATA-sorban) a program a 60050. sorban kilép az alprogramból és a főprogrammal folytatódik.

Amennyiben a régi jel képernyőkódját megtalálta, megállapítja, hogy hányadik oszlopban van és az X3 felveszi az oszlopszám értékét. Ezáltal a PD%-tömb négy index-e (L, X1, X2, X3) rendelkezésre áll. A 60060. sor ennek alapján kikeresi a táblázatból az új jelet és felülírja vele a régit. Rajzoláskor (L=0) ez azt jelenti, hogy a régi jelet egy negyed jellel kiegészíti, törléskor pedig (L=1) egy negyed jellel csökkenti.

Még egy szót a koordinátákról. A koordinátarendszer középpontja a képernyőablak bal alsó sarka, így könnyen ábrázolhatunk a képernyőn függvényeket is.

5.2. Oszlopdiagram

Mérési eredmények, összesítések (mérlegek) grafikus ábrázolásának jól használható és áttekinthető formája az oszlopdiagram.

A vízszintes sorokkal rendkívül jól szemléltethető pl. egy áruház forgalmának vagy egy vállalat termelési eredményének alakulása úgy, hogy minden oszlop az év valamelyik hónapját jellemzi. Sajnos alig van olyan számítógép, amelyik alapvetően képes az ehhez szükséges, megfelelően széles oszlop előállítására. Éppen ezért az alábbiakban közöljük egy ilyen diagram készítésére használható program listáját. Alprogramként, a P8-as programhoz hasonlóan használható.

Most is nagyon előnyösen alkalmazhatjuk a grafikus jeleket. A képernyő és a képernyőjelek felépítéséből adódóan vízszintes irányban 320 különböző vonlahosszúságot tudunk ábrázolni, mivel egy képernyősorban 40 jel fér el minden jel 8 pont szélességű. A 64-es karakterkészletében 0-tól 8-ig (pontban számolva) mindenféle szélességű függőleges oszlopot megtalálunk.

Azt kell csak kiszámítanunk, hogy a kívánt vonalhosszúságot hány egész inverz szóközből és melyik grafikus jelből tudjuk összerakni. Ehhez ismét egy tömböt alkalmazunk.

A program listája:

```
1 REM *** P 9 ***
2 REM
10 DIM XA$(7):FOR I=0 TO 7:READ XA$(I):NEXT
11 REM ------------------------------
12 PRINT"■":Y=160:X=5:V=5
13 Y=Y+1:X=X+2: IF Y=170 THEN END
14 GOSUB 60500:GOTO 13
15 REM ------------------------------
20 DATA " ","▌","▌","▌","▌","■ ","■ ","■ "
60500 YM=320-Y*8:AN$="": IF Y<YM THEN Y=YM
60510 XA=Y/3:G=INT(XA):XA=(XA-G)*8
60520 IF G>0 THEN FOR I=1 TO G:AN$=AN$+"■ ":NEXTI
60530 AN$=AN$+XA$(XA)
60540 C1=PEEK(214):C2=PEEK(211):C3=PEEK(646)
60550 POKE646,SZ:POKE214,X:POKE211,V:SYS58732
60555 PRINT AN$
60560 POKE646,C3:POKE214,C1:POKE211,C2:SYS58732
60565 RETURN
READY.
```

Hogy a program önállóan is futtatható legyen, a 12. és 13. sorokban értékeket adtunk a változóknak. Ha más programokban alprogramként akarjuk használni, ne felejtsük el ezeket törölni! (– a magyar változat készítői)

A 10. és 20. sor jelentése világos, itt történik a szükséges jelek beolvasása. A biztonság kedvéért felsoroljuk a 20. sorban lévő jelek ASCII-kódjait (RVS-ON/RVS-OFF nélkül):

32, 165, 180, 181, 161, 182, 170, 167

Az alprogramba GOSUB 60500 utasítással ugrunk. Az oszlop hosszúságát (0–320) az Y-változóban, a sort (0–24) pedig az X-változóban kell megadnunk.

Grafikák vagy hasonlók kászítésekor azt is megtudhatjuk, hogy az oszlop a képernyőn hol kezdődjön (0–39) (V).

Ha a vonal hossza a felhasználható képernyőtartományt meghaladná, a program 60500. sora azt automatikusan lerövidíti. A 60510. sor kiszámítja, hogy az adott hosszon összesen hány reverz szóköz fér el (G) és hány pont marad (XA).

A 60520. sor a szükséges számú szóközt egy füzérré kapcsolja össze (AN$), amihez 60530. sor utolsóként hozzáfűzi a kiegészítő grafikus jelet.

Hogy ne befolyásoljuk a normál PRINT-utasítást, rögzítjük az aktuális kurzorpozíciót és az éppen használatos színt (C1, C2, C3–60540. sor). A következő sorban a kurzort a megfelelő helyre állítjuk, kívánság szerint megváltoztatjuk a színt (SZ) ás "megrajzoljuk" az oszlopot. Az utolsó sor visszaállítja az eredeti állapotot és befejezi az alprogramot.

Mivel most is kizárólag olyan jeleket használtunk, amelyek a kisbetűs üzemmódban is előállíthatók, ez az alprogram is szinte minden esetben alkalmazható.

5.3. A jelábrázolás üzemmódjai

Ebben a fejezetben tisztázzuk azt a kérdést, hogy az egyes kis- és nagybetűk honnan származnak. Az 1-es szám, ami mondjuk a video-RAM 1024-es tárolórekeszében van, azt határozza meg, hogy a képernyő ennek megfelelő helyén egy A-betű jelenjen meg, de a betű alakjáról semmit sem mond. Az A-betű mintáját (és a többi jelét is, B-től egészen az utolsó grafikus jelig) az ún. jelkészlet – ROM (v. jelgenerátora) tárolja. Ez az 53248–57343 közötti tártartományban található. Ebből a 4 kbyte-ból minden jel 8 byte-ot foglal el, mivel a jelmátrix 8×8 pontból áll.

Minden pontsor 1 byte-nak, így minden bit 1 pontnak felel meg. Ha egy bit értéke 1, akkor a képernyőn a neki megfelelő helyen egy pont jelenik meg, abban a színben, ami a szín–RAM-ban a hozzá tartozó helyen szerepel. Ha a bit 0, akkor a neki megfelelő képernyői pont az 53281. tárolórekeszben rögzített háttérszínben jelenik meg.

A video-RAM-ban rögzített számoknak különleges szerepük van. Ezek

az ún. képernyőkódok. A VIC (Video Interface Chip) ezt 8-cal megszorozza és az így kapott szám adja meg a képernyőkódhoz tartozó jel definíciójának a jelkészlet – ROM-on belüli kezdőcímét. Próbáljuk ki! Válasszunk ki egy tetszőleges jelet, keressük ki a képernyőkódját (nem az ASCII-t!) és indítsuk el a következő kis programot:

```
1 REM *** P 10 ***
2 REM
4 DIM M(7)
5 PRINT"■"
10 INPUT"KEPERNYOKOD:";K
20 AD=53248+K*8
30 POKE 56334,PEEK(56334) AND 254
40 POKE 1,PEEK(1) AND 251
50 FOR I=0 TO 7:M(I)=PEEK(AD+I):NEXT
60 POKE 1,PEEK(1) OR 4
70 POKE 56334,PEEK(56334) OR 1
80 FOR I=0 TO 7:FOR N=7 TO 0 STEP -1
90 IF (M(I) AND 2↑N) THEN PRINT"*";:GOTO 110
100 PRINT".";
110 NEXT N:PRINT:NEXT I
120 PRINT"NYOMJON LE EGY BILLENTYUT !"
130 POKE 198,0:WAIT 198,1:GETA$:GOTO 5
READY.
```

Ezzel a programmal kikereshetjük a nagybetűs üzemmód jeleinek bitmintáit. Ha a kisbetűs üzemmód jeleire is kíváncsiak vagyunk, a 20. sorban a báziscímet 53248-ról 55296-ra kell modosítanunk.

Másik lehetőség: A kiválasztott képernyőkódhoz egyszerűen hozzáadunk 256-ot. (pl: a=1+256=257)

A programban szereplő kis trükkökön nem kell csodálkozni. Ezeket egyelőre még nem ismerjük, de későbbi fejezeteinkben még visszatérünk rájuk. Magyarázatként egyelőre csak annyit, hogy a "programocska" két részből áll. Az első rész (5–70) beolvassa az M(I) tömbbe a jelgenerátor kiválasztott byte-jait. Ehhez kikapcsoljuk a megszakítást (30. sor) és bekapcsoljuk a jelkészlet – ROM-ot (40. sor).

Ezután következik a beolvasást végző FOR ...NEXT ciklus a PEEK-kel (az AD-változó tárolja a kiválasztott jel kezdőcímét), majd az I/O-tartomány és a megszakítás visszakapcsolása.

A második rész szétszedi a bitminta 8 bitjét. Ha az M(I) n-edik bitje 1, akkor az (M(I) AND 2 ↑ n) kifejezés eredménye 1. Ebben az esetben az IF feltétel teljesül, a program a THEN-t követő utasítássorral folytatódik, azaz kivisz a képernyőre egy *-ot. Ellenkező esetben (a bit 0) ide egy pont kerül (100. sor). Az egészben az a különleges, hogy az IF ... THEN utasításban nincsen összehasonlítás, csak egy zárójelben lévő kifejezés.

A program az igen-ágon (THEN utáni utasítássor) fut tovább, ha ennek értéke 0-tól különböző. A kifejezés helyett változó vagy más hasonló is állhat.

De térjünk vissza eredeti témánkhoz. Mint azt a fejezet címe is elárulja, az eddig ismertetett üzemmód nem az egyetlen. A normál üzemódhoz nagyon hasonló az ún. kiterjesztett-szín (Extended Colour) – üzemmód. A jelminta bekapcsolt (1) bitjei továbbra is a szín-RAM-ban meghatározott színű pontokként jelennek meg a képernyőn, a kikapcsolt (0) bitek viszont különböző színűek lehetnek. Színűk a 0–3. (53281–53284) színregiszterből származik. "Aha!" sóhajt fel az Olvasó, hiszen ezeket a CMB-kézikönyv is megemlíti, de rendeltetésükről nem ad bővebb leírást.

Azt, hogy egy kikapcsolt pont a négy szín közül melyikben jelenjen meg, a video-RAM-ban lévő képernyőkód felső két bitjétől függ. Ebben az üzemmódban ezek önálló számként léteznek (szám=7.bit*2+6. bit*1) és a szín származási regiszterének számát adják (pl.: 10=1*2+0=2). Ezt a két bitet természetesen ezután már nem használhatjuk a jelgenerátorra vonatkozó mutatóként, ezért erre a célra most már csak 6 bitünk van. Ez az oka annak, hogy kiterjesztett-szín-üzemmódban összesen csak 64 különböző jelet ábrázolhatunk a képernyőn. Az üzemmód bekapcsolása a

```
POKE 53265, PEEK (53265) OR 64
```

kikapcsolása a

```
POKE 53265, PEEK (53265) AND 191
```

utasítással végezhető.

Most jön a neheze! A többszínű (multicolour) üzemmód viszonylag bonyolult, de helyesen alkalmazva csodás eredményeket érhetünk el vele.

Emlékezzünk csak vissza: minden egyes képernyői jel legfeljebb két színből állhat. Egyik a pontszín (a szín – RAM-ból), a másik a háttérszín (a VIC-regiszterekből). Ezzel szemben többszínű üzemmódban egy jel akár 4 színből is állhat. Ez azonban csak a pont-mátrix egyszerűsítésével érhető el. Ezt az üzemmódot az 53270-es tárrekesz (22-es VIC-regiszter) 4. bitjének bekapcsolásával választhatjuk. Ilyenkor a szín-RAM 3. bitjének rendel-

tetése megváltozik és jelzőként (flag) működik. Amennyiben értéke 0, a jelek továbbra is 8 × 8 pontból épülnek fel, hasonlóan a normál üzemmódhoz. A különbség csak annyi, hogy a 3. bit más célú foglaltsága miatt 16 szín helyett mindössze 8-at (0–7) használhatunk.

Amikor a 3. bit 1, végre a többszínű üzemmód van érvényben. Az eddigi 8 × 8-as pontmátrix-ot egy 4 × 8-as váltja fel, ami azt jelenti, hogy a jelgenerátor minden két bitje jelent 1 pontot a képernyőn. Ha mindkettő 0, a pont háttérszínű.

Még három eset lehetséges:
- ha mindkét bit 1, a VIC a pont színét a szín-RAM-ból,
- ha 0 1, az 1-es,
- ha 1 0, a 2-es háttérszínregiszterből veszi.

Ez az üzemmód BASIC-ből a

```
POKE 53270, PEEK (53270) OR 16
```

utasitássorral kapcsolható be. Kikapcsolása a

```
POKE 53270, PEEK (53270) AND 239
```

utasítássorral végezhető.

A képernyőn megjelenő jelek most eléggé zűrzavarosak, de ha a jelgenerátort átmásoljuk a RAM-ba, ügyes többszínű jeleket tervezhetünk magunknak.

Összefoglalás:

Kiterjesztett-szín-üzemmód be: POKE 53265, PEEK (53265) OR 64
Kiterjesztett-szín-üzemmód ki : POKE 53265, PEEK (53265) AND 191
Többszínű üzemmód be : POKE 53270, PEEK (53270 OR 16
Többszínű üzemmód ki : POKE 53270, PEEK (53270) AND 239

5.4. A jelgenerátor eltolása

Nem nehéz rájönni, hogy a jelgenerátort is a VIC vezérli (ez a C 64-es "fenegyereke"; felügyeli a processzor és a többi építőelem összmunkáját, előállítja a videojelet, vezérli a sprite-okat és a nagyfelbontású grafikát és mellesleg az egész számítógép időzítését ellátja). Bennünket most speciálisan az 53272-es (24-es VIC) regiszter érdekel.

Ezen belül az 1 ... 3 bitek (a jelgenerátor 11–13-as címbitjei) adják meg a jelgenerátor kezdőcímét. Ez még más tényezőktől is függ, de ezek nehezebben befolyásolhatók, ezért a következő adatok inkább a normál konfigurációra vonatkoznak; adott esetben esetleg segédprogramokkal módosíthatók (főleg grafikai segédprogramokra gondolunk).

Az alábbi táblázatban összefoglaltuk, hogy melyik bitkombinációval melyik tartomány címezhető:

000	0
001	2048
010	nagybetűs
011	kisbetűs
100	8192
101	10240
110	12288
111	14336

Speciális eset a 010 és 011 kombináció, mivel ezek ROM-területet címeznek (53248 ill. 55296).

Az 53272-es tárrekesz három bitjéhez csak egy AND és egy OR kapcsolat együttes alkalmazásával férhetünk hozzá.

Az eredeti foglaltság módosításához először is egy AND-del ki kell kapcsolnunk őket: byte AND 1111 0001 (=241). A kérdéses három bitet tehát kikapcsoltuk, a többit nem változtattuk.

Ezután egy OR kapcsolattal beállíthatjuk a kívánt kombinációt. Például: helyezzük át a jelgenerátort a 2048-as címre (BASIC-kezdőcím!). Ehhez az 53272-es tárrekeszben a címbitek 001 kombinációját kell beállítanunk. Képezzük ennek decimális megfelelőjét, ami esetünkben 1. Mivel a byteban ez a három bit eredeti helyiértéke eggyel magasabb, az egészet meg kell szoroznunk 2-vel.

A jelgenerátor áthelyezésének teljes utasítássora:

```
POKE 53272, (PEEK (53272) AND 241) OR 2
```

A jelgenerátort tehát áthelyeztük, de ezzel még nem sokra megyünk. A képernyőn egyelőre pontok összevisszaságát látjuk.

Lássunk munkához. A P 11-es programmal kiolvashatjuk és a RAM területre másolhatjuk a jelgenerátor – ROM-ot. Először azonban át kell helyeznünk a BASIC – kezdőcímet 6144-re, mivel az első 4 k-t most a jelgenerátor fogja igénybe venni. Ez a következőképpen történik:

POKE 43,1:POKE 44,24:POKE 6144,0:CLR

Ha azt akarjuk, hogy a program önműködően áthelyezze a jelgenerátort, egy betöltőprogramot kell alkalmaznunk (ld. 3.3. fejezet). Most viszont már begépelhetjük a következő sorokat:

```
1 REM *** P 11 ***
2 REM
10 POKE56334,PEEK(56334)AND254:REM MEGSZAKITAS KI
20 POKE 1,PEEK(1) AND 251:REM ROM BE
30 FOR I=0 TO 4095:POKE2048+I,PEEK(53248+I):NEXTI
40 POKE 1,PEEK(1) OR 4:REM ROM KI
50 POKE 56334,PEEK(56334) OR 1:REM MEGSZAKITAS BE
READY.
```

Ha most átkapcsolunk az új jelgenerátorra, tapasztalhatjuk, hogy a jelek alakja a régi, de – és ez benne a lényeg – már a RAM-ból származnak és mint tudjuk, itt már a POKE-utasítással könnyen végezhetünk módosításokat. A képernyőjeleket a sprite-okhoz hasonlóan definiálhatjuk, csupán a felépítésükben (nem 21 × 24-es, hanem 8 × 8-as pontmátrix) és a tárolón belül elhelyezkedésükben van különbség.

A jelminták megtekintéséhez ismét felhasználhatjuk a p 10-es programot, de a megszakítás kikapcsolására és az 1. tárolórekesz tartalmának módosítására ebben az esetben nincs szükség, viszont a 20. sorban 2048-ra kell módosítani a báziscímet.

Az új képernyőjeleket egyszerű POKE – utasításokkal írhatjuk be.

Tessék – próbáljuk ki! Különösen többszínű üzemmódban vannak határtalan lehetőségeink.

Az új jelkészletet a

POKE 53272, (PEEK (53272) AND 241) OR 4 (vagy 6)

utasítássorral kapcsolhatjuk ki. (4=nagybetűs, 6=kisbetűs jelkészlet.)

Összefoglalás: A jelgenerátor eltolása

Az 53272-es tárolórekesz 1 ... 3 bitjeit határozzák meg a jelgenerátor kezdőcímét. Ha áthelyezzük a BASIC-tartományba, az általa lefoglalt területet védeni kell (ld. 3.3. fejezet).

A megszakítás kikapcsolása után a jelkészlet-ROM kiolvasható és a RAM-ba másolható.

Az egyedi jelkészlet bekapcsolása:

```
POKE 53272, (PEEK (53272) AND 241) OR X
```

Kikapcsolása:

```
POKE 53272, (PEEK (53272) AND 241) OR 4 (vagy 6)
```

5.5. A video-RAM eltolása

Az 53272-es tárrekesz jelgenerátorhoz hasonlóan a video-RAM is áthelyezhető. Ez esetben a 4 ... 7 biteket használjuk.

Ezzel a 4 bittel a tv-RAM-ot a tártartományon belül 1 K-s lépésként tolhatjuk el. Normál esetben csak a 4. bit értéke 1, ami az 1024 ... 2023 tártartományt jelöli ki.

Az alábbi táblázatban összefoglaljuk a bitkombinációk és tárcímek összefüggését:

0000	0
0001	1024
0010	2048
0011	3072
0100	ROM
0101	ROM
0110	ROM
0111	ROM
1000	8192
1001	9216
1010	10240
1011	11264
1100	12288
1101	13312
1110	14336
1111	15360

Amint látjuk, a ROM-mal jelölt kombinációk kivételelt képeznek. Erre azért van szükség, hogy a VIC hozzáférhessen a ROM-ban lévő jelgenerátorhoz. Ez a tartomány "áttükröződik" a 4096 ... 8191 területre. A VIC számára tehát a jelkészlet-ROM itt található és nem az 53248-as címen! Az átkapcsolás ismét az AND, OR, PEEK és POKE utasításokkal végezhető. Először is törölni kell a négy felső bitet And 15-tel. Ezután át kell alakítani a bináris kombinációt decimálissá. Ha pl. a tv-RAM-ot a 15360-as tárcímre akarjuk elhelyezni, akkor ez 15 lesz.

Ezt a számot a bitek valódi helyiértéke miatt meg kell szorozni 16-tal, majd az így kapott eredményt OR-ral a tártartalomhoz adjuk. A teljes utasítássor így néz ki:

POKE 53272, (PEEK(53272) AND 15) OR X

Nincs valami feltűnő azon a számon, amit a bitkombinációból kiszámítottunk? Bizony, van. Ez a szám pontosan azt adja meg, hogy a video-RAM a tártartomágy hányadik kbyte-jába kerüljön. Ezután nem is kell foglalkoznunk a bitkombináció átszámításával, hanem egyszerűen behelyettesíthetjük a kiválasztott kbyte számát a következő utasítássorba.

POKE 53272, (PEEK(53272) AND 15) OR K*16

Ha ezt így ahogy van, kipróbáljuk, csalódni fogunk. A képernyőn a jelek teljes összevisszaságban jelennek meg, és a billentyűzet hatástalan (kivéve a RUN-STOP) RESTORE, SHIFT/C=, valamint a SYS 63 738 utasítást (a RETURN-t előtte is és utána is le kell nyomni) – a magyar változat készítőinek megjegyzése.) Elfelejtettük közölni az operációs rendszerrel a video-RAM új helyét. A VIC most olyan helyről szerzi a képernyőre vonatkozó információkat, ahová az operációs rendszer még nem írt semmit. Ha ebben a helyzetben lenyomjuk a CLR-gombot, az operációs rendszer még a régi képernyőtárat és a szín-RAM-ot is törli. Azért erre a bajra is van orvosság.

A 648-as tárrekesz közli a számítógéppel a video-RAM báziscímének felső byte-ját. Ezt úgy kapjuk, hogy a báziscímet elosztjuk 256-tal. Példánknál maradva: 15360/256=60, azaz mindent helyrehozhatunk egy

POKE 648,60

utasítással. A C 64-es ismét rendesen működik.

Szerencsére itt is van egy egyszerűbb módszer. Elegendő megszoroznunk a kbyte-számot (itt 15) 4-gyel, máris megkapjuk a keresett felső byte-ot.

Még valami, amit a sprite-ok használatakor figyelembe kell venni. A sprite-mutatókat (sprite-definició kezdőcíme) már nem a 2040 ... 2047 tártartományban találjuk, hanem – mivel ez a tv-RAM felső 8 byte-ja – ez is eltolódott a 16376 ... 16383 tartományba.

Ne felejtkezzünk el aról, hogy a video-RAM által lefoglalt BASIC-tárolót védenünk kell.

Csábító lehetőség, hogy két különböző képernyőoldalt definiálhatunk és tetszés szerint váltogathatjuk azokat. Igaz, a BASIC PRINT – utasításai csak az éppen bekapcsolt képernyőoldalra vonatkoznak, a másik oldalhoz továbbra is csak a PEEK- és POKE-utasításokkal férhetünk hozzá. A másik probléma: mivel a szín-RAM nem tolható el, mindkét képernyőoldalon ugyanazokat a színeket kell használnunk.

Összefoglalás: A video-RAM áthelyezése

A tv-RAM-ot az 53272-es tárolórekesz 4 ... 7 bitjei vezérlik. Táron belüli áthelyezése az alábbi utasítássorral végezhető:

POOKE 53272, (PEEK(53272) AND 15) OR K*16

Az operációs rendszer számára a

POKE 648, K*4

utasítássorral adjuk meg a video-RAM új kezdőcímét.

A K mindkét esetben a képernyőtárat átvevő kilobyte számát jelenti.

5.6. Néhány trükk a képernyőn

Normál szöveg-üzemmódban is létezik néhány olyan trükk, amivel a programozást, a kivitelt és egyebeket megkönnyíthetjük.

Kezdjük a színekkel. A képernyőjelek pillanatnyilag érvényes színét a 646-os rekesz tárolja. A POKE 646, színkód utasítással ugyanazt az eredményt érjük el, mint a CTRL+szín ill. Commodore+szín billentyű-kombinációval. Ennek az eljárásnak az az előnye, hogy a színkódok közvetlenül adhatók meg. Többek közt akkor használható, ha pl. a színek átkapcsolását egy RND-érték függvényében véletlenszerűvé akarjuk tenni.

A 647-es rekesz a kurzor alatti színt tárolja (akkor is, ha az ki van kapcsolva). Sajnos, ez a POKE 647, X utasítással nem módosítható.

Apropó módosítás: Minden színt tároló rekeszre érvényes, hogy 4 ... 7 bitjei módosíthatók. Ez a tény azonban nem sokat jelent, mivel a 0 ... 15 színkódok előállításához csak a 0–3 biteket használjuk. Éppen ezért ne csodálkozzunk, ha pl. az 55296-os rekeszben – amit mondjuk előzőleg nullákkal töltöttünk fel – váratlanul egy 32-es vagy egyéb érték jelenik meg.

A 243/244-es rekeszekben található az aktuális kurzorpozícó szín-RAM-ra vonatkozó mutatója. Az operációs rendszer ezt mindig akkor kérdezi le, amikor egy jel kivitelre kerül.

Vezérlőjelek esetében a mutató nem változik, mert ilyenkor az utasítás végrehajtásához nincs szükség a szín-RAM-ra. Pl. PRINT "(HOME)"-nál a mutató nem változik, míg PRINT "(HOME) ABC"-nél felveszi az új kurzorpozíciónak megfelelő értéket.

Hasonló mutató a video-RAM-ra vonatkozóan is létezik. Két részből áll és a 209/210-es rekeszekben található. Arra a tárcímre mutat, ahol az aktuális (azaz a kurzort tartalmazó) képernyősor kezdődik. Ehhez az értékhez hozzáadva a 211-es rekeszben lévő aktuális oszlopszámot (0–39), megkapjuk a kurzor video-RAM-on belüli helyzetét.

Az aktuális sor száma (0–24) a 214-es rekeszben van. E két byte (211/214) segítségével a kurzor képernyői pozicionálása egyszerűen megoldható úgy, hogy a kiválasztott sor és oszlop számát POKE-kal a megfelelő rekeszbe írjuk. Ez azonban még nem elég, mert az operációs rendszer ebből még nem tudja, hogy odébb kell vinnie a kurzort. Ezt a feladatot egy ROM-rutin végzi el, amit a SYS 58732 utasítással indítunk. A teljes utasítássor tehát a következő:

POKE 211, oszlop: POKE 214, sor: SYS 58732

Ezt a trükköt a P 9 programban már alkalmaztuk (5.2. alfejezet).

Ugye jó lenne, ha GET-bevitelkor is láthatnánk a kurzort? Nos, ez sem lehetetlen! Azt, hogy a kurzor megjelenjen-e vagy sem a 204-es rekesz határozza meg. Ha tartalma 0 (azaz PEEK (204) eredménye 0), a kurzor látható, ha 1, nem látható. Amikor elindítunk egy programot, az értelmező egy 1-es tölt a 204-es rekeszbe, ezáltal eltűnik a kurzor.

Amit az operációs rendszer tud, ara mi is képesek vagyunk. A POKE 204,0 utasítással "menet közben" egyszerűen bekapcsolhatjuk a kurzort. A megszakító-rutinnak eszébe sincs, hogy emiatt reklamáljon. Csak arra ügyeljünk, hogy nehogy éppen munka közben kapcsoljuk ki újra (POKE 204,1-gyel), mert ilyenkor inverz karakterek maradhatnak a képernyőn. De itt is segít egy POKE! A 207-es regiszter adja meg azt, hogy a kurzor éppen a bekapcsolási vagy a kikapcsolási fázisban (reverz v. normal ábrázolás) van-e. Kikapcsolási fázisban értéke 0, bekapcsoláskor 1. A

POKE 207,0:POKE 204,1

utasítássorral tehát a kurzort a legmegfelelőbb üzemi állapotban kapcsolhatjuk ki.

A bevitelnél maradva bemutatunk még egy INPUT-utasítással kapcsolatos, tippet: az

INPUT "szöveg (kurzor jobbra) (kurzor jobbra) Z (kurzor balra) (kurzor balra) (kurzor balra)"; A$

utasítássorral a Z-betűt a bevitelnél kurzorjelként használhatjuk, ami egy billentyű lenyomásakor felülíródik.

Az idézőjelek közt természetesen a megfelelő kurzorvezérlő jelnek kell szerepelnie, zárójel nélkül.

A következő utasítás a reverz-üzemmódra vonatkozik. Függetlenül attól, hogy a kivitelre szánt füzér tartalmazza-e a megfelelő vezérlőjelet vagy sem, a

POKE 199,1

utasítás bekapcsolja a reverz ábrázolási módot. Kikapcsolása

POKE 199,0

utasítással végezhető.

Szeretnénk a füzér vezérlőjelét a program által képernyőre vinni? Kérem, semmi akadálya. Rendelkezésünkre áll a 216-os tárrekesz, ami megadja a kintlévő inzertek számát. Tudjuk, hogy inzert-üzemmódban a gép nem hajtja végre a vezérlőjeleket, csupán inverz jelként kijelzi őket a képernyőn. Ezt az üzemmódot a

POKE 216, X

utasítással kapcsoljuk be, ahol X>0

. Végül játszunk még egyszer "operációs rendszert". Aki dolgozott már szalagegységgel, az tudja, hogy amikor a kazettásegység operációs rendszere dolgozik, kikapcsolódik a képernyő. Ennek is a VIC az oka. Az 53265-ös tárolórekesz 4. bitje határozza meg, hogy a képernyő látható-e vagy sem. A

POKE 53265, PEEK (53265) AND 239

utasítással kikapcsolhatjuk, a

POKE 53265, PEEK (53265) OR 16

utasítással pedig újból visszakapcsolhatjuk.

Jelszín módosítása	: POKE 646, színkód
Aktuális jelszín	: PRINT PEEK (647)
Aktuális pozíció a szín-RAM-ban	: PRINT PEEK (243)+256*PEEK(244)
Aktuális pozíció a video-RAM-ban:	PRINT PEEK (209)+256*PEEK(210)+ +PEEK(211)
A kurzor oszlopa	: PRINT PEEK (211)
A kurzor sora	: PRINT PEEK (214)
A kurzor beállítása	: POKE 211,
oszlopszám	: POKE 214,
sorszám	: SYS 58372
Kurzor be	: POKE 204,0
Kurzor ki	: POKE 207,0: POKE 204, 1
Speciális beviteli kurzorjel	: INPUT "szöveg (2*kurzor jobbra) jel (3*kurzor balra)"; A$
Reverz üzemmód be	: POKE 199,1
Reverz üzemmód ki	: POKE 199,0
Inzert üzemmód be	: POKE 216,X
Képernyő kikapcsolása	: POKE 53265, PEEK (53265) AND 239
Képernyő bekapcsolása	: POKE 53265, PEEK (53265) OR 16

(lásd: Melléklet: : M 24; M 25 programok)

6. A NAGYFELBONTÁSÚ GRAFIKA

"Most jön a hal mellé a vaj" (ahogy ezt Tegtmeier mondaná). Feltárjuk a nagyfelbontású grafikát, amit a Commodore 64-es tervezői igen alaposan elrejtettek az operációs rendszer mögé. Mint ebben a könyvben már megszoktuk, a célravezető "Szezám nyílj ki!" most is a POKE – utasítás.

6.1. A grafikus üzemmód

A normál szöveg-üzemmódhoz hasonlóan a grafikus ábrázolásnak is különböző módjai vannak. Hiányzik azonban a kiterjesztett – szín- (extended colour-) üzemmód, aminek a nagyfelbontású ábrázolásban úgysem volna értelme. Normál üzemmódban 320×200, azaz 64000 pontot ábrázolhatunk, ami a jelgenerátorhoz hasonlóan 8000 byte-ban fér el. Ezt a nyolcszor nagyobb képernyőtárat hívják bit-térképnek (bit-map). Mint egy valódi térképen az 1-es bitek itt is azt adják meg, hogy a valóságban (itt képernyő) van-e domb (itt pont) vagy nincs. A pontok színét a video-RAM tárolja (nem a szín – RAM!). A valamikori képernyőtár minden byte-ja egy 8×8 pontból álló tartományt határoz meg, ami szöveg-üzemmódban 1 jel lenne. A négy felső bit adja a tartományon belül a bekapcsolt pontok (bit=1), a négy alsó pedig a kikapcsolt pontok (bit=0), azaz a háttér színét.

Többszínű (de most nagyfelbontású!) üzemmódban a pontmátrixot ismét egyszerűsítenünk kell. A 320×200 pont helyett ebben az esetben csak 160×200 pontunk van. Mivel most is 2–2 bit határoz meg 1 képernyői pontot, a bittérkép számára 8000 byte szükséges, de ezzel minden 8×8 (ill. 4×8 képernyői ponthoz 4 színt adhatunk meg. A színek az egykori video-RAM mellett a 0. háttérszínregiszterből (53281) és a szín-RAM-ból származnak, mégpedig a következő hozzárendelés szerint: 00=0. háttérszínregiszter; 01= video-RAM alsó 4 bit; 10=video-RAM felső 4 bit, 11=szín-RAM.

6.2. A bit-térkép

Mindenekelőtt valamit a bit-térkép tárón belüli elhelyezkedéséről. Most is, mint már sokszor, az 53272-es regiszter bitjei játszanak meghatározó szerepet. A bit-térkép a 3. bit állapotától függően a 8192. (3. bit=1) ill. a 0. tárrekesztől kezdődően helyezkedhet el. Ez utóbbi számunkra kevésbé használható, mivel itt helyezkedik el a nullás lap is, amit az operációs rendszer tudta és engedélye nélkül nem írhatunk át. A bit-térkép felépítése a jelgenerátoréhoz hasonló. Az első 8 byte az első jelblokk 8 pontsorát adja, ...stb. Emiatt előfordulhat, hogy a grafika bekapcsolásakor normál képernyőjelek jelennek meg a monitoron. Másoljuk csak át a jelgenerátort a bit-térkép területére és módosítsunk néhány byte-ot – az eredmény ismerősnek fog tűnni.

Többszínű üzemmódban is hasonló a helyzet, de itt 2–2 bit jelent 1–1 dupla szélességű képernyői pontot (ezt már az 5. fejezetből ismerjük).

A bit-térkép elhelyezkedése miatt elengedhetetlen a BASIC-tár védelme. Mivel tárigénye csak 8000 byte és nem 8192 (=8 kbyte), a tárat a 16192-es rekesztől használhatjuk. A mutatók kiszámítása már ismert (3.3 fejezet).

Mivel a normál BASIC-tár a 2048-as címtől kezdődik, tulajdonképpen a 6 kbyte-unk megmarad. Ezeket például a sprite-ok számára vehetjük igénybe. További kilobyte-ok kellenek még a színek és a további képernyőoldalak létrehozásához. Már volt szó róla, hogy (visszaélve a gép adta lehetőségekkel) a video-RAM-ot színtárként is felhasználjuk. Ilyenkor persze felülíródik az eredeti szöveg-képernyő. Ezt úgy kerülhetjük el, hogy a nagyfelbontású grafika bekapcsolása előtt egy másik képernyőoldalra kapcsolunk át (ld. 5.5. fejezet). A kétféle üzemmód így a továbbiakban már nem zavarja egymást. Egyetlen hátránya, hogy ha a sprite-okat mindkét üzemmódban használni akarjuk, kétszer kell "bePOKE-olnunk" a sprite-mutatókat: egyszer a szöveg-üzemmód számára a 2040 ... 2047-es, egyszer pedig a nagyfelbontású üzemmód számára az áthelyezett tartományba. Sajnos, többszínű üzemmódban ez nem valami elegánsan működik, mert mivel itt is a szín-RAM-ot használjuk, az eredeti szöveg színe módosulhat.

6.3. A grafikus üzemmód bekapcsolása

A grafikus üzemmód bekapcsolása három lépésben történik. Először védelem alá kell helyeznünk a bit-térképet. Ez (ha a hozzátartozó program már kész) egy betöltőprogrammal elvégezhető, amelynek az alábbi utasításokat kell tartalmaznia:

POKE 43, 65: POKE 44, 63: POKE 16192, O: CLR

A program összeállításakor ezeket előzőleg közvetlen üzemmódban kell bevinnünk. A grafikát ezután a programon belül kapcsolhatjuk be, mégpedig az 53265-ös tárolórekesz 5. bitjének bekapcsolásával. A VIC ebből tudja, hogy nem jeleket, hanem nagyfelbontású grafikát ábrázolunk. Ha többszínű grafikát akarunk készíteni, ezenkívül be kell kapcsolnunk még az 53270-es tárrekesz 4. bitjét is (ugyanúgy, mint szöveg-üzemmódban). Ez azonban még nem elég. Az 53272-es regiszter 3. bitjével meg kell adnunk a bit-térkép helyzetét, azaz ezt is be kell kapcsolnunk, végül át kell helyeznünk a video-RAM-ot, hogy a képernyőtartalmat ne tegyük tönkre.

Ha mindezt végrehajtottuk, a képernyőn egy rettenetes zűrzavar keletkezik. Hátra van még a harmadik lépés! Egy FOR-NEXT ciklussal törölnünk kell a bit-térképet, valamint a video-RAM-ot. Kész!

A teljes utasítássor a következő:

```
1 REM *** P 12 ***
2 REM
10 POKE 53265,59:REM GRAFIKUS UZEMMOD BE
20 REM POKE 53270,216 (TOBBSZINU UZEMMOD BE)
30 REM BIT-TERKEP 8192-RE;VIDEORAM 1024-RE
35 POKE 53272,24
40 REM BIT-TERKEP TORLESE
45 FOR I=8192 TO 16191:POKE I,0:NEXT
50 REM PONT LILA/HATTER FEHER
55 FOR I=1024 TO 2023:POKE I,4*16+1:NEXT
60 WAIT198,1:REM VARAKOZAS EGY BILLENTYURE
READY.
```

Ezután a video-RAM-ot a 2048 ... 3047 tartományban találjuk. A 3072-es tárcímtől további 5 k áll rendelkezésünkre, amit különböző célokra, pl.

sprite-ok gépi kódú rutinok ...stb. számára használhatunk. Mivel minden grafika véget ér egyszer, következzenek a kikapcsolást végző POKE-utasítások:

POKE 53265, 155: REM grafikus üzemmód kikapcsolása
(POKE 53270, 8: REM többszínű üzemmód kikapcsolása)
POKE 53272, 21: REM nagybetűs jelkészlet bekapcsolása

Kipróbáltuk első grafikánkat? Ha igen, biztosan bosszant a bit-képkép törlésének lassúsága. Módosítsuk a programunkat az alábbiak szerint:

```
1 REM *** P 13 ***
2 REM
10 REM * GEPI RUTIN *
20 REM ***************
30 FOR I= 3600 TO 3659:READ A:POKE I,A:NEXT
40 DATA 169,32,133,252,169,0,133,251,162,31
45 DATA 160,0,145,251,136,208,251,230,252,202
50 DATA 208,246,160,64,145,251,136,16,251,169
55 DATA 8,133,252,165,2,162,3,160,0,145,251
60 DATA 136,208,251,230,252,202,208,246,160
70 DATA 232,145,251,136,208,251,141,0,11,96
80 REM ***************
90 REM
110 POKE 53265,59
120 POKE 53272,40
130 SYS 3600
READY.
```

Amint látjuk, a bit-térkép törlését ás a video-RAM feltöltését most egy gépi rutin végzi, amit a megfelelő helyen SYS 3600-zal indítunk. Azt, hogy a video-RAM-ba milyen színek kerülnek, a 2-es tárrekesz tartalmától függ és a

POKE 2, pontszín * 16 + háttérszín

utasítással adhatjuk meg. Mivel a programból ez kimaradt, mindkét színkód 0, tehát fekete.

A rutin nem helyhezkötött relocatible), azaz akár a kazetta-pufferben akár máshol elhelyezhető. Kezdőcíme mindig az a byte, amivel a 30. sor-

ban a FOR ... NEXT ciklus kezdődik. Próbáljuk ki és nézzük meg a futási sebességét. Az eddigi hosszadalmas művelet a másodperc töredéke alatt lezajlik.

Végül még egy apró ötlet: ha a sprite-okat, a grafikai oldalt, a színeket és a törlőprogramot a főprogrammal együtt akarjuk lemezre vagy kazettára rögzíteni, akkor ez viszonylag egyszerű.

Ha mindennel elkészültünk és a sprite-ok, a grafikus oldal és a gépi program már a védett tartományban van, állítsuk vissza a BASIC-mutatót a 43/44-es rekeszben a normál BASIC-kezdőcímre és utána sima SAVE-utasítással tároljuk az egészet.

Természetesen a mutatót egy magasabb címre is állíthatjuk, ha pl. a szín-RAM-ot nem akarjuk rögzíteni. Ezáltal tárkapacitást takaríthatunk meg. Ezután a programot egy

LOAD"név",8,1

utasítással betöltve a tárban készen rendelkezésre állnak a sprite-ok, grafika ... stb. (A betöltéshez betöltőprogramot kell használnunk, ami beállítja a mutatókat.) Mindez főként játékprogramoknál hasznos.

Összefoglalás: A grafika bekapcsolása

A következő POKE – utasításokkal kapcsolhatjuk be a nagyfelbontású ill. a többszínű grafikát. Ekkor a video-RAM a 2048 ... 3047 tártartományban van, a BASIC-kezdőcímet pedig 16192-re kell felemelni.

POKE 53265, 59: REM a nagyfelbontású grafika bekapcsolása
(POKE 53270, 216: REM a többszínű üzemmód bekapcsolása)
POKE 53272, 40: REM a bit-térkép és a video-RAM eltolása

Ezután töröljük a video-RAM-ot és a bit-térképet.

A grafika kikapcsolása:

POKE 53265, 155: REM grafika kikapcsolása
(POKE 53270, 8: REM a többszínű üzemmód kikapcsolása)
POKE 53272, 21: REM nagybetű/grafikus jelkészlet bekapcsolása

6.4. Pont rajzolása

Mielőtt egy pontot akarnánk elhelyezni a grafikus képernyőn, célszerű az egész képet milliméterpapíron megtervezni, majd ebből kiszámítani a byte-ok értékeit. Ha ezután rengeteg munka, gyötrődés és dühkitörések sora rémlik fel előttünk, akkor ez nem is olyan rendkívüli. Elkerülésük végett közlünk két olyan rutint, amelyekkel a grafikai programozás lényegesen megkönnyíthető.

6.4.1. Pont rajzolása a nagyfelbontású képernyőn

Az alábbiakban a programlista elvileg ugyanúgy működik, mint az 5.1. fejezetben található negyedpont grafika rutin (P 8).

Mivel azonban most nem kell speciális grafikus jeleket beírnunk, nincs szükség a pontkoordinátákat átalakító táblázatra sem.

```
1 REM *** P 14 ***
2 REM
61000 REM PONT RAJZOLASA ES TORLESE
61001 REM ------------------------
61010 Y=199-Y: IF Y<0 OR Y>199 THEN RETURN
61020 IF X<0 OR X>319 THEN RETURN
61030 X1=INT(X/8):X2=INT(Y/8)
61035 AD=8192+8*X1+320*X2+(YAND7)
61040 X3=2↑(7-(XAND7)):CA=1024+X1+40*X2
61050 POKECA,(PEEK(CA)AND15)OR16*SZ
61060 IFL=1THENPOKEAD,PEEK(AD)AND(255-X3):RETURN
61070 POKEAD,PEEK(AD)ORX3:RETURN
READY.
```

Alprogramként alkalmazva a GOSUB 61000 utasítással hívhatjuk le. A pont helyzetét az X(0–319) és Y(0–199) koordinátákkal adjuk meg. A koordinátarendszer középpontja a bal alsó sarok. Ha a megadott X vagy Y érték a lehetséges tartományon kívülre esik, az alprogram a 61010/61020. sorban befejeződik. Így pl. egy vonalat látszólag a képernyőhatárokon túlra is húzhatunk.

A 61030. sor kiszámítja annak a byte-nak a címét, amelyiket a bit-

térképen belül módosítani kell. Az INT(X/8) a szín-RAM-ban lefoglalt oszlopot adja. Ezt az értéket nyolccal meg kell szorozni, mivel a szín-RAM egy rekesze a bit-térkép 8 byte-ját képviseli. Hasonlóan az INT (Y/8) a szín-RAM-ban elfoglalt sor számát adja. Ahhoz, hogy a bit-térképen belüli sornak megfelelő címet megkapjuk, ezt az értéket meg kell szoroznunk a soronkénti max. pontszámmal, azaz 320-szal. Az Y AND 7 kifejezés végül megadja, hogy a pont a színnégyzeten belül melyik sorban van.

Az X3 változó értékét össze kell kapcsolnunk a már meghatározott bitekkel aszerint, hogy a hozzátartozó pontot törölni vagy rajzolni akarjuk-e. Végül AD tartalmazza a pont, CA a szín tércímét. A 61050. sor a megfelelő rekesz felső négy bitjébe beírja (POKE) az SZ változóban megadott színt (0–15).

Vegyük figyelembe, hogy ezáltal a teljes négyzet (8 × 8 pont) színe megváltozik!

L = 1 az alprogram számára azt jelenti, hogy a kérdéses pontot törölni kell. Ebben az esetben a program a 61060. sorba ágazik el, egyébként az utolsó sorban lévő "rajzolás" rész kerül végrehajtásra.

Az alprogram alkalmazásának tipikus példái a Melléklet M 15-ös és M 16-os programjai.

6.4.2. Pontok többszínű üzemmódban

Többszínű üzemmódban pontonként két bitet kell, a színtől függően különböző kombinációban, bekapcsolnunk. Ez a módszer azt feltételezi, hogy pontonként (pontosabban 2-pontonként) kétszer kell lefuttatni a pont elhelyezését végző rutint. Az első bit részére egyszerűen megduplázzuk az X-koordinátát, majd ehhez 1-et hozzáadva megkapjuk a másodikat. Mivel ez a két bit minden esetben ugyanabban a byte-ban van, a POKE-cím kiszámításának kétszeres lehívásától eltekinthetünk.

Figyeljük meg:

```
1 REM *** P 14 - 1 ***
2 REM
61000 REM TOBBSZINU PONT RAJZOLASA
61001 REM ------------------------
61010 Y=199-Y: IF Y<0 OR Y>199 THEN RETURN
61020 X=2*X: IF X<0 OR X>318 THEN RETURN
61030 X1=INT(X/8):X2=INT(Y/8)
```

```
61035 AD=8192+8*X1+320*X2+(YAND7)
61040 X3=2↑(7-(XAND7)):X4=2↑(7-((X+1)AND7))
61050 POKEAD,PEEK(AD)AND(255-(X3+X4))
61060 POKEAD,PEEK(AD)OR((SZAND1)*X3+(SZAND2)/2*X4)
61065 RETURN
READY.
```

Erre a programra alprogramként GOSUB 61000-rel hivatkozhatunk. A koordinátáját ismét az X(0–159) és Y(0–199) változókban adjuk meg. A színeket és a törlés- vagy rajzolás-üzemmódot már nem kell megadnunk, ezt jelenti a színszám (0–3) az SZ változóban, amit a bitkombináció ad meg. Ha a pontot törölni kell, az SZ-nek egyszerűen 0 értéket adunk. A bitkombinációból adódó színeket adott esetben előzőleg POKE-kal a megfelelő regiszterbe be kell tölteni (a video-RAM esetében ezt a törlő-rutin végzi). A 61010 ...61030 sorok egy apró változtatástól eltekintve a normál nagyfelbontású üzemmód programjával (P 14) azonosak.

A 61040-es sor kiszámítja az összekapcsolási maszkból a két bitet (X3, X4), majd a 61050-es sor törli ezeket. Végül a 61060. sor a bitkombinációt beírja a megfelelő byte-ba. Az SZ AND 1 összefüggés a kombináció alsó, az (SZ AND 2)/2 a felső byte-ját adja. Ha az egyik bit 0, az egész szorzat értéke 0. A tár megfelelő bitje OR 0 kapcsolat eredménye mindig a bit eredeti állapota, míg a bit OR 1 kapcsolaté mindig 1.
Egy tipikus alkalmazási példa:

```
X= 100:Y= 50 : REM koordináták
SZ= 2        : Rem szín a video-RAM felső bitjeiből
GOSUB 61000  : REM ugrás az alprogramba
```

6.5. Egyenes rajzolása

A következő alprogram mindkét grafikus üzemmódban alkalmazható. A pontok elhelyezésére felhasználhatjuk az előző fejezet rutinjait (P 14, P 14–1) ezért ez a program itt csak a koordináták kiszámítását végzi:

```
1 REM *** P 15 ***
2 REM
61100 REM EGYENES RAJZOLASA
61101 REM ------------------
61110 IF ABS(XE-XA)<ABS(YE-YA)THEN 61160
61120 SP=(YE-YA)/ABS(XE-XA+1E-20):YK=YA
61130 FOR XX=XA TO XE STEP SGN(XE-XA)
61140 YK=YK+SP:Y=INT(YK+.5):X=XX:GOSUB61000
61150 NEXTXX:RETURN
61160 SP=(XE-XA)/ABS(YE-YA+1E-20):XK=XA
61170 FOR XX=YA TO YE STEP SGN(YE-YA)
61180 XK=XK+SP:X=INT(XK+.5):Y=XX:GOSUB 61000
61190 NEXTXX:RETURN
READY.
```

Az alprogram lehívása GOSUB 61100-zal lehetséges. Az egyenes kezdőpontjának koordinátái XA, YA, végpontjáé XE, YE. Ezen kívül a pontok elhelyezéséhez meg kell még adnunk a színt (SZ) és az üzemmódot (L-rajzolás/törlés) ill. többszínű üzemmódban a színszámot (SZ).

Maga az algoritmus tulajdonképpen nagyon egyszerű. Először is megvizsgáljuk, hogy az X-koordináták közti távolság kisebb-e, mint az Y-koordináták távolsága (61110. sor).

Ha igen, az egész folyamat megfordul, de a működési mód ugyanaz. Ennek magyarázata a következő:

Tételezzük fel, hogy az X-koordináták távolsága nagyobb, mint az Y-koordinátáké. Ez azt jelenti, hogy az egyenes több pontjának kell azonos Y-koordinátával rendelkeznie, a képernyőn ugyanis sohasem húzhatunk tökéletes ferde vonalat, csakis egy megközelítő cikk-cakk-mintát. Ez a képernyő felépítéséből adódik. "Letapogatjuk" tehát, egy FOR ... NEXT ciklussal XA-tól XE-ig az összes X-koordinátát és mindegyikhez kiszámítjuk a megfelelő Y-t (61130. sor). Fordított esetben, azaz ha az X-koordináták különbsége kisebb, mint az Y-koordinátáké, előfordulhat, hogy egy X-koordinátához több Y-t kell rendelni.

Ekkor a fordított eljárással az Y-koordinátákat tapogatjuk le és kiszámítjuk a megfelelő X-értékeket. Ez a számítás is nagyon egyszerű. A ciklus indítása előtt kiszámítjuk a léptéket (SP). Ez nem más, mint két egymás melletti pont Y-irányú távolsága. A ciklus minden átfutásakor ennyivel nő az YK-segédváltozó. Ennek kerekített értéke lesz a pont valódi Y-koordinátája: Y=INT (YK + .5). Ha ez is megvan, a program elágazik a 61000. sorban kezdődő alprogramba (P 14 vagy P 14–1), ami elhelyezi a pontot a képernyőn (61140. sor). Ha a pont a megengedett tartományon kívülre esik, a rajzoló alprogram befejeződik.

Sajnos a program nem túl gyors, de egyszerű feladatokra (pl. függvényábrázolás) tökéletesen megfelel. Amikor a gyorsaság is lényeges, speciális segédprogramokhoz kell folyamodnunk, mint amilyen pl. a DATA BECKER gondozásában megjelent SUPERGRAPHIK 64. Ez a könyv sok olyan utasítást tartalmaz, ami lényegesen meggyorsítja a spriteokkal, nagyfelbontású grafikával ... stb. a munkát.

6.6. Kör rajzolása

Sok más alakzat mellett a kör az egyik leggyakrabban alkalmazott grafikus elem. Nem építhető fel egy vonalrendszerből, ezért az alábbiakban közlünk egy erre alkalmas szubrutint. Igaz, rendkívül lassú, de még mindig jobb, ha egy kört lassan rajzolunk meg, mint ha teljesen nélkülöznünk kell. A P 15-ös programhoz hasonlóan ez is mindkét grafikus üzemmódban alkalmazható.

```
1 REM *** P 16 ***
2 REM
61200 REM KOR RAJZOLASA
61201 REM ------------
61210 FOR XX=0 TO R*0.7
61220 YY=INT(SQR(1-(XX/R)↑2)*R)
61230 X=XA+XX:Y=YA+YY:GOSUB61000
61240 X=XA+XX:Y=YA-YY:GOSUB61000
61250 X=XA-XX:Y=YA-YY:GOSUB61000
61260 X=XA-XX:Y=YA+YY:GOSUB61000
61270 X=XA+YY:Y=YA+XX:GOSUB61000
61280 X=XA+YY:Y=YA-XX:GOSUB61000
61290 X=XA-YY:Y=YA-XX:GOSUB61000
61300 X=XA-YY:Y=YA+XX:GOSUB61000
61310 NEXTXX:RETURN
READY.
```

Lehívása: GOSUB 61200. Megadandók a kör középpontjának koordinátái (XA, YA), a kör képernyőpontokban mért sugara (R), a szín (SZ) és az üzemmód (L-törlés/rajzolás) ill. többszínű üzemmódban a színszám (SZ).

A rajzoláshoz megint a 6.4. fejezet alprogramjait (P 14, P 14–1) vesszük igénybe.

A program működési elvét megértjük, ha megfigyeljük, hogyan keletkezik a kör a képernyőn. Kiindulópontunk a kör egyenlete: $X \uparrow 2 + Y \uparrow 2 = 1$, vagy másképpen: Y=SQR (1 − $X \uparrow 2$).

Egy FOR ... NEXT ciklussal kiszámítjuk a kb. 45°-os középponti szöghöz tartozó ív pontjainak minden X-koordinátához tartozó Y-koordinátáját. Ha a ciklust a 90°-os középponti szögig ismételnénk, ugyanazzal a problémával kerülnénk szembe, mint amit az egyenes rajzolásakor tapasztaltunk. Minél meredekebben esik lefelé a körív, annál gyakoribb, hogy egy X-értékhez több Y tartozik.

Emiatt a kör további pontjait úgy rajzoljuk meg, hogy az így kialakuló negyedkört a 61230 ... 61300 sorokkal a hiányzó helyekre tükrözzük. Ha a középpont X-koordinátájából levonjuk a sugár aktuális pozíciójának X-koordinátáját, a kör baloldali részét kapjuk ... stb.

Mivel a fent megadott egyenlet csak az ún. egységnyi sugarú (R=1) körre érvényes, a számításban (61220. sor) az X-koordinátát először el kell osztani, majd az egészet ismét meg kell szorozni a sugár értékével.

Az M 16 program tetszőleges kört rajzol a nagyfelbontású képernyőre. (Melléklet.)

7. A SPRITE-OK

A 64-es legismertebb jellemzői a sprite-ok. Nincs még ilyen sokoldalú grafikai módszer. Talán éppen ezért – az összes grafikai lehetőség közül –, a CBM-kézikönyv is csak ezt említi meg. Ám a Commodore itt is nagyon rejtélyes és ismét reménytelen zűrzavart okoz.

Hol van a sprite-ok ütközésvizsgálatának magyarázata?

Mik azok a többszínű sprite-ok?

Az általuk kínált különböző lehetőségek részletes leírását tartalmazzák a következő fejezetek. Ezek ismeretében már kifoghatunk a Commodore-on!

7.1. A többszínű sprite-ok

Igen, jól olvastuk. A normál, nagyfelbontású "minigrafika" mellett a VIC többszínű sprite-ok programozására is alkalmas.

A többszínű üzemmód kiválasztása a VIC 28-as regisztere (53276 a megfelelő tárcím) a sprite számától függő, bitjének bekapcsolásával történik.

Ha például a 6-os számú sprite-ot akarjuk többszínűként definiálni, a következő utasítássort kell alkalmaznunk:

```
POKE 53276, PEEK (53276) OR (2 ↑ 6)
```

Természetesen a következő utasítássorral ez a bit újból nullázható.

```
POKE 53276, PEEK (53276) AND (255-2 ↑ 6)
```

Ezzel a sprite-ot máris átkapcsoltuk többszínű üzemmódra. Mivel most ismét a mátrix 2 bitje határoz meg 1 pontot, összesen 12×21 pontunk marad. Azt már tudjuk, hogy ez az üzemmód, hogy működik, már csak arra vagyunk kíváncsiak, melyik bitkombináció melyik színt eredményezi.

Ha mindkét bit 0, az illető pont átlátszó, ami azt jelenti, hogy ebben a pontban a háttér (pl. egy betű) fog látszani. Egyébként a pont színe 01 bitkombináció esetén a 37-es, 11 esetén a 38-as VIC-regiszterből (53285/53286 – többszín-regiszterek), 10 esetén pedig a normál sprite-szín regiszterből (VIC 39–46) származik.

Ez utóbbi sprite-onként különböző lehet, de a többi (az ún. multicolour színek) nem. Ezek minden sprite-nál ugyanabból a regiszterből származnak és színkódjuk csak 0 ...7 lehet. A többszínű sprite-okat ugyanúgy definiáljuk, mint az egyszínűeket, csupán a bitek és pontok egymáshoz rendelése más. A képernyőkoordináták sem változnak. Legnagyobb előny kétségtelenül az, hogy a sprite- és grafikus-üzemmód összekeverhető. Ezáltal egyszerre és egymás mellett ábrázolhatunk a képernyőn nagyfelbontású grafikát és többszínű sprite-okat. A VIC-nek teljesen mindegy, hogy a képernyőn éppen grafika vagy más jel van-e.

Egyetlen korlátozás: a lemezegység működtetése alatt ki kell kapcsolnunk a sprite-okat (POKE 53269,0), mivel a VIC az ütemezést is vezérli és ez annál lassabb, minél több sprite van a képernyőn. Ez zavarhatja az adatátvitelt.

Összefoglalás: Többszínű sprite-ok

Üzemmód bekapcsolása: A sprite számának megfelelő bit bekapcsolása a 28-as VIC-regiszterben (53276).

A sprite színei a következő regiszterekből származnak:

VIC 37	(01 bitkombináció)	
VIC 38	(11 bitkombináció)	
VIC 39 ...46	(10 bitkombináció)	normál sprite-szín regiszterek

00 bitkombináció esetén a pont "átlátszó".

A sprite-ok és a különböző grafikai üzemmódok összekeverhetők.

7.2. A sprite-ok ütköztetése

A VIC regisztereiben minden egyes sprite-sprite és sprite-háttérjel közti ütközés rögzítésre kerül. Ha az ütközés két vagy több sprite között jön létre, a 30-as VIC regiszter (53278) az illetékes. Az esemény bekövetkeztét a részt vevő sprite-ok számainak megfelelő bitek bekapcsolásával jelzi. A

PRINT PEEK (53278) AND 2 ↑ *N*

utasítássorral kérdezhető le, hogy az N-edik sprite ütközött-e vagy sem. Ha igen, a fenti összefüggés eredménye: 2 ↑ *N*, ha nem: 0. Két sprite találkozása

csak akkor minősül ütközésnek, ha két valódi pontjuk érintkezett és nem üres területeikkel fedik egymást. Az ütközést jelző bitek mindaddig bekapcsolva maradnak, míg a

POKE 53278, 0

utasítással nem töröljük őket. Előfordulhat tehát, hogy a regiszter még akkor is ütközést jelez, mikor a sprite-ok már rég elhagyták egymást. Éppen ezért ajánlatos az ütközés lekérdezése után közvetlenül a törlést is végrehajtani. Ha a nullázás és a lekérdezés között újbóli ütközés történt, a megfelelő bitek azonnal bekapcsolódnak és a következő PEEK – utasítás már ezt állapítja meg.

A sprite – háttérjel közti ütközés ellenőrzése azonos módon végezhető. A sprite számának megfelelő bit 1-re vált, ha a bit-térképen vagy a jelgenerátorban a sprite egy 1-essel jelzett (bekapcsolt) pontot fed. Ezt a 31-es VIC-regiszter (53279) jegyzi.

A sprite-ok ütközésvizsgálatának programozását az alábbi kis játékprogrammal szemléltetjük. Egy nagyon egyszerű autóversenyről van szó. Célja az, hogy a Z (= balra) és /(= jobbra) billentyűkkel kikerüljük az autó útjába kerülő akadályokat.

Az autó más autókkal vagy az út szélével való érintkezése az ütközési regiszterben feljegyzésre kerül és karambolhoz vezet. A program bevitelekor természetesen elhagyhatjuk a REM-es sorokat, hiszen ezek csak lassítanák a játékot.

Összefoglalás: Ütközések

A sprite – sprite közti ütközést a 30-as, a sprite – háttérjel közti ütközést pedig a 31-es VIC – regiszter rögzíti, mégpedig az ütközésben részt vevő sprite-ok számainak megfelelő bitek bekapcsolásával.

A bitek mindaddig ebben az állapotban maradnak, míg a felhasználó a megfelelő utasítással nem törli őket.

```
1 REM *** P 17 ***
2 REM
3 REM * AUTOVERSENY *
4 REM
10 REM KEPERNYO ELOKESZITESE
15 PRINT"■":POKE53280,0:POKE53281,0:V=53248
```

```
20 REM AUTO-SPRITE BEOLVASASA
25 FOR I=832 TO 894:READ A:POKEI,A:NEXT
30 REM UTKOZES-SPRITE BEOLVASASA
35 FOR I=896 TO 958:READ A:POKEI,A:NEXT
40 REM SPRITE-MUTATO ES SZINEK
45 POKE2040,13:POKE2041,13:POKEV+39,1:POKEV+40,2
50 FORI=0TO24:POKE1036+I*40,160:POKE55308+I*40,1
60 REM UT A KEPERNYORE
65 POKE1051+I*40,160:POKE55323+I*40,1:NEXTI
70 REM STARTHELYZET
75 POKEV,168:POKEV+1,170:POKEV+21,3:X=168
80 REM AKADALY STARTHELYZETE
85 POKE V+2,168:POKE V+3,0:HX=168:HY=0
90 POKEV+30,0:POKEV+31,0:REM UTK.REG.-EK TORL.-E
100 A=PEEK(203):REM BILLENTYUZET LEKERDEZESE
110 IF A=12 THEN X=X-1:REM Z-BILLENTYU
120 IF A=55 THEN X=X+1:REM /-BILLENTYU
130 POKE V,X:REM AUTO MOZGATASA
140 IFPEEK(V+30)<>0ORPEEK(V+31)<>0THENPOKE2040,14:
    FORI=0TO500:NEXT:RUN:REM UTK.
150 HY=HY+2:IF HY=>240 THEN HY=30:REM MOZGATAS LE
160 HX=HX+INT(RND(TI)*5)-2:REM KOOR.& BAL SZEL
165 IF HX<120 THEN HX=120
170 IF HX>216 THEN HX=216:REM JOBB SZEL
180 POKE V+2,HX:POKE V+3,HY:GOTO 100
1000 REM AUTO-SPRITE
1100 DATA   0,  0,  0
1101 DATA   0,126,  0
1102 DATA   0,126,  0
1103 DATA   0,255,  0
1104 DATA  12,255, 48
1105 DATA  15,255,240
1106 DATA  12,255, 48
1107 DATA   0,255,  0
1108 DATA   0,255,  0
1109 DATA   1,231,128
1110 DATA   1,195,128
1111 DATA   1,195,128
1112 DATA   1,195,128
```

```
1113 DATA   3,135,192
1114 DATA   3,135,192
1115 DATA 115,255,206
1116 DATA 115,255,206
1117 DATA 127,255,254
1118 DATA 115,255,206
1119 DATA 115,255,206
1120 DATA   0,  0,  0
2000 REM UTKOZES-SPRITE
2100 DATA 123, 20,  0
2101 DATA   0, 24,  0
2102 DATA  30, 44, 77
2103 DATA  21,126,  3
2104 DATA 240,125, 48
2105 DATA  15,205,240
2106 DATA  22,155, 41
2107 DATA   1,205,156
2108 DATA   0,155,  0
2109 DATA   1,201,108
2110 DATA   1,105,108
2111 DATA   1, 95, 28
2112 DATA   1,155,158
2113 DATA  23,115,192
2114 DATA  32,242,132
2115 DATA  35,239,216
2116 DATA   1, 95, 28
2117 DATA  32,242,132
2118 DATA  68,155,  0
2119 DATA  27,125, 48
2120 DATA   0,  0,  0

READY.

READY.
```

7.3. Elsőbbség és mozgási tartomány

A képernyőjelek, grafikák és sprite-ok kölcsönös átfedésére különböző lehetőségeink vannak. Alapesetben a sprite-ok az aktuális képernyőtartalom előtt jelennek meg. Gyakran ennek ellenkezőjét szeretnénk, pl hogy a repülő egy ház mögött mozogjon. A VIC-nek erre a célra is van egy regisztere, mégpedig a 27-es (53275= V + 27, ahol V= a VIC báziscíme = 53248). Ennek mindegyik bitje egy sprite-ot képvisel. Ha egy bit értéke 1, a hozzátartozó sprite a képernyőjelek mögött jelenik meg. Alapesetben mindegyik bit 0, tehát a sprite-oknak van elsőbbségük a képernyőjelekkel szemben.

Különleges jelenség, amikor több sprite-ot ábrázolunk eltérő elsőbbségi joggal. Mint tudjuk, a legkisebb számú sprite mindig a többi előtt jelenik meg. Ha most a háttérnek elsőbbséget adunk ezzel a sprite-tal szemben, akkor ez ezentúl a háttérjelek mögött, de még mindig a többi sprite előtt fog megjelenni, akkor is, ha ezek a háttérjelek előtt vannak.

Ezzel optikai csalódásokat idézhetünk elő a képernyőn.

Megpróbálkoztunk már egy grafika és egy sprite megjelenítésével összehangolásával? Ha igen, akkor már megállapíthattuk, hogy koordinátarendszerünk nem azonos. A sprite-ok mozgási tartománya nagyobb, hogy kiúszhassanak a képernyőből.

Egy ilyen "grafika a grafikában" a képernyő bal felső sarkában a (24, 50) koordinátákkal rendelkezik. Ez a két szám a korrekciós tényező szerepét tölti be, amelyeket a grafikai koordinátákhoz hozzáadva kapjuk meg a sprite helyes pozícióját. A képernyő közepén tehát koordinátái 160+24 és 100+50. A vonatkozási pont mindig a sprite bal felső sarka.

Összefoglalás: Elsőbbség és mozgási tartomány

Egy sprite elsőbbsége (háttérjelek előtt/mögött) a 27-es VIC-regiszter (53275) hozzátartozó bitjének állapotától függ. A bit bekapcsolásával a sprite a háttérjelek mögé kerül.

A grafikus koordinátarendszerben a sprite-ok helyzetének meghatározásához korrekciós tényezőkkel kell számolnunk, amelyeket a grafikus koordinátákhoz hozzá kell adnunk. Ezek: X-irányban 24, Y-irányban 50.

7.4. Ötletek a sprite-ok programozásához

Sok játékprogramban alkalmazzák az ún. animációt, pl. ahhoz, hogy természethűen ábrázolhassák ahogy egy kis sprite-emberke egy sprite-lány után fut. Ez a képernyőn úgy látszik, mintha kezét, lábát külön mozgatná. Amikor aztán jobban megnézzük, észrevesszük, hogy mind a lábnak, mind pedig a kéznek csupán két vagy három különböző helyzete van. Ezzel az animáció elvét már meg is értettük. A sprite ebben az esetben két külön blokkból áll, amelyeket a mozgás folyamán felváltva ki- és bekapcsolunk. Az egyik blokkban van mondjuk az emberke zárt karokkal és lábakkal, a másikban pedig ugyanez az emberke a mozgás másik fázisában, pl. kinyújtott karral és felemelt lábbal. A sprite-mutató (2040 ... 2047) segítségével ezt a két képet felváltva jelenítjük meg a képernyőn és ezáltal egy futó figura képét hozzuk létre.

A valóságban csak a sprite "mintáját" cseréljük ki. Ilyen egyszerű. Előfeltétele természetesen az, hogy a különböző képekhez elegendő tárkapacitás álljon rendelkezésre. Ilyenkor szintén célszerű felemelni a BASIC-kezdőcímet. Ha nagyfelbontású grafikát alkalmazunk, mindenképpen elég helyünk van.

Gyakran szükséges vagy legalábbis hasznos, hogy a sprite-blokkokat mágneslemezre vagy kazettára rögzítsük. Egy erre való programot már a 4.1. fejezetben ismertettünk.

Érdekes lehet az az elképzelés, hogy nagyfelbontású sprite-okat kis grafikus képernyőként a szöveg-képernyőn alkalmazzunk.

Tételezzük fel, hogy egy tetszőleges függvénygörbét szeretnénk a nagyfelbontású képernyőn ábrázolni, egyidejűleg néhány megjegyzéssel is kiegészítve. Egyik lehetőségünk az, hogy a szükséges betű bitmintáját beolvassuk a jelgenerátorból és a megfelelő grafikus pontokból mesterségesen létrehozzuk a jelet. Ez azonban elég körülményes és ugyanígy a fordított eljárás is, amikor a jelkészletet úgy definiáljuk "át", hogy a grafikonhoz szükséges pontok bekerüljenek a jelmátrixba. Maradnak a sprite-ok. Vegyünk belőlük négyet és helyezzük el ezeket úgy a képernyőn, hogy a kívánt helyen négyzetet alkossanak. Ezután számítsuk ki a pontokhoz a sprite-mátrixon belüli biteket.

Ezt elvégzi helyettünk az alábbi rutin. Közelebbi magyarázatra nincs szükség, mivel felépítésében és működésében megfelel a már ismertetett, nagyfelbontású grafikus pont kijelzésére írt programnak (P 14).

```
1 REM *** P 18 ***
2 REM
5 PRINT"■"
10 FOR I=704 TO 767:POKE I,0:NEXT
20 FOR I=832 TO 1023:POKE I,0:NEXT
30 POKE 2040,11:POKE 2041,13
35 POKE 2042,14:POKE 2043,15
40 V=53248
45 POKEV,100:POKEV+1,100:POKEV+2,148:POKEV+3,100
50 POKEV+4,100:POKEV+5,142:POKEV+6,148:POKEV+7,142
60 FOR I=39 TO 42:POKEV+I,1:NEXTI
65 POKEV+21,15:POKEV+23,15:POKEV+29,15
100 INPUT"■X-KOORD.";X:INPUT"Y-KOORD.";Y
110 GOSUB 62000:GOTO 100
62000 Y=41-Y:IF Y<0 OR Y>41 THEN RETURN
62010 IF X<0 OR X>47 THEN RETURN
62020 BX=INT(X/24):BY=INT(Y/21)
62025 IF BX=0 AND BY=0 THEN BA=704:GOTO 62040
62030 BA=768+BX*64+BY*128
62040 BX=X-24*BX:BY=Y-21*BY
62050 X1=INT(BX/8):X2=7-(BXAND7):X3=BY*3
62060 AD=BA+X1+X3
62070 IFL=1THENPOKEAD,PEEK(AD)AND(255-2↑X2):RETURN
62080 POKE AD,PEEK(AD)OR(2↑X2):RETURN
READY.
```

A nagyfelbontású grafikához hasonlóan a koordinátákat az X és Y, az üzemmódot (törlés/rajzolás) pedig az L változóba töltjük. A 0 ... 3-as sprite-okat és a 11-, 13-, 14-, 15-ös blokkokat használjuk. A sprite-okat mindkét irányban nagyítjuk. Eredeti nagyságukban is megjeleníthetők, de akkor a pozíciót helyesbíteni kell (30./40. sor).

A 4 sprite most összesen 48 × 42 pontból áll. Mivel nagyfelbontásúak, minden pont színe azonos. Ezt a 60. sorban határozzuk meg.

Aki akarja, átírhatja a vonalrajzoló rutint sprite-grafikára, igazán nem nehéz. Minden jellegzetes funkció (elsőbbség, ütközés ... stb.) hagyományos módon alkalmazható a 0 ... 3 sprite-okra, mivel az alprogram csak a 704–766, 832–894, 896–958 és 960–1022 tartományokra hat.

Érdemes egy kicsit kísérletezni!

Ezzel a sprite-ok programozását tárgyaló fejezetünk végére értünk, de ez nem jelenti azt, hogy nem kísérletezhetünk tovább a VIC regisztereivel. Bizonyára még sok felfedezésre váró terület létezik, különösen a sprite-ok tekintetében.

8. A HANGELŐÁLLÍTÁS

Ami a képernyő számára a VIC, ugyanaz a hang terén a SID (Sound Interface Device). Ebben minden hangzásfajtára egy regiszter van. Sajnos, a CBM-kézikönyv ezt sem említi. Mivel a hangelőállítás összes lehetőségének ismertetése túlságosan terjedelmes lenne, itt csak a programozás alapjaira térünk ki.[1]

8.1. A SID működése

Ebben a fejezetben elsősorban arról lesz szó, hogy milyen folyamatok játszódnak le a számítógépben a hangelőállítás közben.

Ha az ún. startbitet 1-re állítjuk, a SID először is megállapítja, hogy milyen frekvenciájúnak kell lennie a hangnak. Ezután előállítja a megfelelő rezgést. Ez bekerül a hullámforrás – modulátorba, ahol elnyeri a beprogramozott hullámformát (négyszög, háromszög, fűrészfog, zaj) és a rá jellemző hangképet.

Ezután a SID meghatározza a hang burkológörbéjét. Ez nem más, mint a hang megszólalása és elhallgatása közti időszak különböző fázisaira jellemző hangerő. A burkológörbéhez 4 paraméter megadása szükséges. Az első a felfutás (attack), ami azt adja meg, hogy a megszólaló hang milyen gyorsan éri el a (valamelyik regiszterben) beállított maximális hangerőt. Ezt követi a lecsengés (decay) fázisa, amikor a hangerő meghatározott idő alatt visszaesik egy alacsonyabb értékre. Ezt az értéket a kitartás (sustain) fázis jellemzi, ami a másik három paramétertől eltérően nem időtartamot, hanem a maximumhoz viszonyított hangerőt jelenti. A 4. fázis a kioltás (release), ami a hang kikapcsolásának sebességét adja. Ezzel visszhangot, utórezgést állíthatunk elő.

A négyszöghullámoknál ezenkívül még az impulzus/szünet viszonyt is megadhatjuk, amivel a bekapcsolási- és kikapcsolási impulzus arányát szabályozhatjuk.

---

[1] További ismeretek megszerzésére a Data Becker – NOVOTRADE – 1986 Zenekönyv c. könyvét ajánljuk.

További lehetőségek (amikre it nem akarunk bővebben kitérni) pl. a gyűrűs-moduláció, amelynél az egyik szólam hangja a két másik szólamtól függ, és a szűrők, amelyekkel különböző frekvenciatartományok szűrhetők ki.

8.2. A programozás

Most jön a lényeg. Megismerkedünk a hangok és hangsorozatok programozásával. Részben eltérünk majd a CBM-kézikönyvben leírt eljárástól, mivel az a lehetőségek egy részének kihasználását egyszerűen lehetetlenné teszi.

Először is tudnunk kell, hogy a SID báziscíme: 54272. A SID-regiszterek programozásakor ezt mindig hozzá kell adnunk az illetékes SID-regiszter számához. A hangokat 3 szólamban tudjuk előállítani. A zárójelben szereplő regiszterszámok a 2. és 3. szólamra vonatkoznak. Akárhogy nézzen is ki a programunk, van valami, aminek az elején mindig szerepelnie kell: ez a hangerő. Ezt a 24-es SID – regiszter (54296) alsó négy bitjével kell beállítanunk.

A PEEK-kel való kiolvasására irányuló kísérlet teljesen felesleges, ugyanis ez és 0-tól 24-ig az összes SID-regiszter csak írható, de nem olvasható. A PEEK – utasítás teljesen értelmetlen eredményt ad.

Ennek éppen a fordítottja igaz a 25 ... 28 regiszterekre. Ezek kiolvashatók, de a POKE – utasítás velük szemben hatástalan.

De térjünk vissza a zenéhez. Mivel a hangerőregiszter alsó négy bitje alapállapotban 0, a kívánt számot egyszerű POKE-utasítással beírhatjuk. A

```
POKE 54296, 0
```

utasítás a hang kikapcsolását, a

```
POKE 54296, 15
```

pedig a maximális hangerő beállítását jelenti. Mindhárom szólam hangerejét egyszerre adjuk meg.

Következik a frekvencia, vagyis a hang magassága.

Összesen 65536 különböző frekvenciából választhatunk, teljesen tetszőlegesen. Dallamok programozásánál nagyon jól használható a kézikönyv függelékében lévő hangjegytáblázat.

Azt, hogy a frekvenciaszám miként bontható alsó- és felső byte-ra, a mutatókról szóló fejezetből már ismerjük.

Ezt a számot az 1. szólam részére a 0/1, a 2. szólam részére a 7/8, a 3. szólam részére a 14/15 regiszterekbe kell POKE-utasítással beírnunk.

Ezután jön a burkológörbe meghatározása. A felfutási (A) és hanyatlási (D) fázis hosszát az 5. (ill. 17. és 19.) regiszterben állítjuk be, mégpedig az előbbit a felső (4 ... 7), utóbbit az alsó (0 ... 3) bitekben. A kitartási (S) és kioltási (R) fázist hasonló módon a 6. ill. (13. és 20.) regiszterben rögzíthetjük. A kitartási fázis hangerejét a felső, a kioltási fázis időtartamát az alsó bitek képviselik. Ha a felső bitek értéke 0, a szólam néma marad. Nullától eltérő érték a 24-es regiszterben beállított maximálishoz viszonyított hangerőt adja meg. Négyszöghullám esetén a SID-nek az impulzus/szünet arányra is szükség van. Ennek értéke 0 ... 4095 lehet és a 2/3, 9/10 ill. 16/17 regiszterpárokban kell rögzíteni. A felső byte-oknak csak az alsó (0 ... 3) bitjeit használjuk. Ezekben a regiszterekben tehát 15-nél nagyobb számokat nem lehet megkülönböztetni.

Nos, eddig úgy haladtunk, mint a kézikönyv. A hullámforma kiválasztásához a 4. (ill. 11. és 18.) regisztert használjuk. Néhány VIC-regiszterhez hasonlóan ebben is minden bitnek sajátos jelentése van. A 0. bit, mint már említettük, az ún. start–stop bit. Ha értéke 1, a megfelelő szólam bekapcsolódik, a hang a burkológörbének megfelelően megszólal és a kitartási fázis szintjén addig szól, míg a bit értéke 0 nem lesz. Ekkor a 6-os regiszterben beállított idő alatt a hang befejeződik, a hangerő 0-ra csökken. A programozáskor tudnunk kell, hogy a SID a kioltási fázisban, ugyanabban a szólamban nem tud új hangot előállítani. Ha pl. olyan dallamot akarunk programozni, amelyben a hangok gyors egymásutánban követik egymást, akkor ezt úgy érhetjük el, hogy viszonylag rövid kioltási időket választunk. Az 1-es a szinkronizáció, a 2-es a gyűrűsmoduláció bitje. Ezekről bővebb információt a szakirodalomból szerezhetünk. A 3-as bit ismét nagyon jól használható. Ha egyidejűleg több hullámformát akarunk bekapcsolni, előfordulhat, hogy a SID leblokkol, azaz egyáltalán nem állít elő több hangot. A 3-as bit bekapcsolásával és a hullámformák törlésével a SID "rövidzárlata" feloldható. Ehhez tehát a

```
POKE 54276, 8
```

utasítást használjuk.

Végül a 4 ... 7 bitek a hullámforma kiválasztására alkalmasak a következők szerint: 4-es bit – háromszöghullám, 5-ös bit – fűrészfoghullám, 6-os

bit – négyszöghullám, 7-es bit – zaj. Ezek keverhetők is egymással. A hang előállításához a start-bitet és a hullámforma bitjét egyszerre kell bekapcsolni. Ebből adódnak a CBM-kézikönyvben közölt, különböző hangszínekre vonatkozó kódok (17, 33, 65, 129). A kikapcsolás viszont nem végezhető a 4. (ill. 11. és 18.) regiszter egyszerű nullázásával. Ez olyan lenne, mintha az autópálya közepén hirtelen leállítanánk a motort. A SID egyszerre csak nem találna adatokat a hullámformára vonatkozóan és a burkológörbét sem tudná rendesen befejezni. Az eredmény a tipikus kikapcsolási kattanás lenne. Ha csak a 0. bitet kapcsoljuk ki, elmarad a kattanás és a hang lágyan kicseng.

Az eljárási mód szemléltetésére és a C 64-es kézikönyvében található, nem futtatható lista helyett bemutatunk egy olyan programot, ami lehetővé teszi, hogy a billentyűzeten dallamot játsszunk:

```
1 REM *** P 19 ***
2 REM
10 PRINT"■"
20 PRINT" W E   T Y U"
30 PRINT"A S D F G H J K"
100 S=54272
110 POKE S+24,15:REM HANGERO
120 POKE S+5,136:REM FELFUTAS ES HANYATLAS
130 POKE S+6,248:REM KITARTAS ES KIOLTAS
140 POKE S+4,8:REM SID INICIALIZALAS
150 FORI=0TO40:NEXTI:POKES+4,16:REM STARTBIT=0
160 GET A$:IF A$="" THEN 160
170 IF A$="A" THEN POKES,207:POKES+1,34
180 IF A$="S" THEN POKES, 18:POKES+1,39
190 IF A$="D" THEN POKES,219:POKES+1,43
200 IF A$="F" THEN POKES,118:POKES+1,46
210 IF A$="G" THEN POKES, 39:POKES+1,52
220 IF A$="H" THEN POKES,138:POKES+1,58
230 IF A$="J" THEN POKES,181:POKES+1,65
240 IF A$="K" THEN POKES,157:POKES+1,69
250 IF A$="W" THEN POKES,255:POKES+1,36
260 IF A$="E" THEN POKES,101:POKES+1,41
270 IF A$="T" THEN POKES, 58:POKES+1,49
280 IF A$="Y" THEN POKES, 65:POKES+1,55
290 IF A$="U" THEN POKES,  5:POKES+1,62
300 POKE S+4,17:GOTO 150
READY.
```

A 10.–30. sor jelzi a billentyűk foglaltságát. Ezután következik az előkészítő rész (100–140). A 150. sorban egy kis várakozási ciklus található, ami időt biztosít a hang lecsengésének. Ezenkívül itt állítjuk be a start–stop-bitet a POKE-utasítással 0-ra. A hullámformát itt és a 300. sorban módosíthatjuk.

A 160 ... 290. sorok funkciója világos; itt rendeljük hozzá a billentyűkhöz a frekvenciát, azaz a hangmagasságot.

A hangelőállítással kapcsolatos bővebb információkért a speciális irodalomhoz kell fordulnunk, mint pl. a DATA BECKER – NOVOTRADE kiadásában (1986) megjelent Zenekönyv. Ebben különleges programozástechnikai megoldásokkal is találkozhatunk.

Összefoglalás: Hangelőállítás

Hangerősség beállítása a 24-es regiszterben.

Értéktartománya:	0–15.
Felfutási idő:	az 5/12/19 regiszterek felső négy bitje
Felfutási idő:	az 5/12/19 regiszterek felső négy bitje
Lecsengési idő:	az 5/12/19 regiszterek alsó négy bitje
Kitartás erőssége:	a 6/13/20 regiszterek felső négy bitje
Kioltási idő:	a 6/13/20 regiszterek alsó négy bitje
Hullámforma:	4/11/18 regiszter
	4. bit – háromszöghullám
	5. bit – fűrészfoghullám
	6. bit – négyszöghullám
	7. bit – zaj
Ezenkívül:	3. bit – a SID inicializálása
	1. bit – start – stop bit
Frekvencia:	0/1, 7/8 vagy 14/15 regiszterpárok

9. A BILLENTYŰZET

Külsőleg ítélve a C 64-es legfeltűnőbb eleme a billentyűzet. Minősége, az öszes többi hasonló árkategóriába tartozó számítógépéhez viszonyítva, a legjobb. Ebben a fejezetben meggyőződhetünk arról, hogy nem csak kényelmes programozásra alkalmas, hanem bizonyos programozási trükköket is lehetővé tesz.

9.1. A billentyűzet felépítése és működése

Kezdjük a billentyűzet lekérdezésével. Ezt egyszerű esetben BASIC – programból az INPUT- és a GET-utasításokkal végezzük. A CBM – kézikönyv ezenkívül elárulja, hogy 0 készülékszámon OPEN – utasítással is hozzáférhető. Ebben az esetben a lemezegység és kazettás egység használata során már megismert utasításokat alkalmazzuk. Ilyenkor azonban, a normál INPUT – utasítástól eltérően, nem jelenik meg kérdőjel a képernyőn. Emiatt jogosan gondolhatunk arra, hogy a billentyűzet valamilyen interface-en keresztül csatlakoztatva van az I/O-tartományhoz. Ez az interface a CIA 1.

A lekérdezés a két párhuzamos porton (ami felhasználását tekintve nagyon hasonlít a user-porthoz) végezhető. Ehhez a 64 billentyű elektronikusan 8 sorba és 8 oszlopba van elhelyezve (4. ábra). A két port közül az egyiket kivitelre programozták. Itt a lekérdezett oszlop kiviteli értékei jelennek meg. Ha egy billentyűt lenyomunk, akkor ezt a bevitelre kapcsolt második port regisztrálja. A megszakítási rutinnak tehát nincs más dolga, mint a 8 oszlopot sorra megvizsgálni és megállapítani a lenyomott billentyűt. A ROM-ban lévő dekódertáblázat alapján ezután kiszámításra kerül a billentyű ASCII-kódja és ez ideiglenesen a billentyűzetpufferbe kerül. Amikor az értelmező közvetlen üzemmódban dolgozik, előveszi és végrehajtja a pufferben tárolt ASCII-kódot (ha ez pl. 13=RETURN, végrehajt egy utasítást).

Egy BASIC-program futása közben a billentyűzetpuffer mindaddig változatlan marad, míg elő nem fordul egy GET- vagy INPUT-utasítás, vagy be nem fejeződik a program.

Get-utasítás esetén az értelmező a puffertárolóból előveszi az első jelet és átadja azt az utasításban megadott változónak. Az INPUT hasonlóan

								56321	Bit
1	£	+	9	7	5	3	INST DEL	254	0
←	*	P	I	Y	R	W	RET	253	1
CNTRL	;	L	J	G	D	A	⇐ CRSR ⇒	251	2
2	CLR HOME	−	Ø	8	6	4	F7	247	3
SPACE	SHIFT jobbra	.	M	B	C	Z	F1	239	4
C=	=	:	K	H	F	S	F3	223	5
Q	↑	@	O	U	T	E	F5	191	6
RUN STOP	/	,	N	V	X	SHIFT balra	⇑ CRSR ⇓	127	7

56320	127	191	223	239	247	251	253	254
Bit	7	6	5	4	3	2	1	0

SHIFT LOCK (= bal SHIFT) és RESTORE nem lekérdezhető.

4. ábra

működik, de ebben az esetben a jel a képernyőn is megjelenik és a lekérdezés mindaddig ismétlődik, míg a RETURN-t le nem nyomjuk.

A leírt billentyűzet mátrixnak két érdekessége van. Az egyik, hogy nem tartalmazza a RESTORE-gombot. Ez ugyanis közvetlenül hat a processzorra (hasonlóan a RESET-hez, ld. 1.6. fejezet) és ott egy speciális megszakítást vált ki. Ez a rutin ellenőrzi azt, hogy egyidejűleg nem nyomtuk-e le a RUN/STOP-billentyűt is. Ha igen, a gép egyfajta mini-RESET-et hajt végre, különben minden rendesen fut tovább.

A másik a SHIFT-billentyűkkel kapcsolatos. Mivel a baloldali és a jobboldali SHIFT-billentyű különböző oszlopban van, a számítógép különbséget tud tenni köztük. Ezzel szemben a SHIFT-LOCK-billentyű csak a baloldali SHIFT egy különleges változata, ami azt jelenti, hogy ezt a két billentyűt nem tudja egymástól megkülönböztetni.

Mindez a VC–20-asra is vonatkozik, csupán a billentyűk elektronikus elrendezésében van eltérés.

9.2. Két billentyű egyidejű lekérdezése

Most az előző fejezet ismereteit átküldjük a gyakorlatba.

Sok program megkívánja, hogy egyidejűleg több billentyűt is lekérdezhessünk, pl. hogy két űrhajót egymástól függetlenül irányíthassunk. Nézzük meg mégegyszer a billentyűzetmátrix-ot (4. ábra).

A billentyűzet lekérdezés két tárolója az 56320-as és 56321-es. Alapesetben minden bitjük 1. Egy oszlop kiválasztásához az 56320-as regiszter megfelelő bitjét 0-ra kell állítanunk. Hasonlóan zajlik a visszajelzés is. Ha egy billentyűt lenyomunk, az 56321-es regiszter hozzá tartozó bitje 0-ra vált.

Már régóta tudjuk, mire képes a megszakító-rutin. Egy POKE-utasítással kiválaszthatjuk valamelyik oszlopot és ellenőrizhetjük a lekérdezendő billentyű bitjeit. Ha a két billentyű más-más oszlopban van, akkor ezeket egyszerűen egymás után kérdezzük le. Mivel a megszakító-rutin zavaró lehet, egyszerűen kapcsoljuk ki.

Ezenkívül egy

```
POKE 788, 52
```

utasítással a RUN-STOP-billentyűt is ki kell iktatnunk, mert enélkül a legalsó sor lekérdezése megszakítást idézne elő.

Mindent összevetve a következő utasítássor adódik:

```
POKE 56334, PEEK (56334) AND 254
POKE 788, 52
POKE 56320, OSZLOPKÓD
IF(PEEK (56321) AND (2↑BITSZÁM)=0 THEN PRINT"BILL.
                                    LENYOMVA"
POKE 56334, PEEK (56334) OR 1
POKE 788,49
```

Ezzel a kis programmal tehát egy billentyű kérdezhető le. Ha egyszerre többet akarunk megfigyelni, további IF ... THEN szerkezetekre és az oszlop kiválasztásához további POKE-utasításokra van szükség.

Az oszlopkódot a kód=255 – 2↑-os (os-oszlopszám) összefüggésből számítjuk ki. Az oszlopszámon az oszlop kiválasztására szolgáló bit byte-on belüli helyzetét értjük. Az IF ... THEN-szerkezet feladata annak ellenőrzése, hogy a kiválasztott bit 0, vagy nem.

A sor- és oszlopszám a 4. ábrából állapítható meg.

Két billentyű egyidejű lekérdezésére a 653-as tárolórekesz egy másik lehetőséget kínál. Ez a rekesz rögzíti, hogy a SHIFT-, COMMODORE-, vagy CTRL-billentyűket lenyomtuk-e. A SHIFT-billentyű a 0. bitet, a COMMODORE az 1. bitet és a CTRL a 2. bitet kapcsolja be. A bitek bekapcsolása egymástól független tehát ha mindhárom billentyűt egyszerre lenyomjuk, mindhárom bit értéke egyszerre lesz 1.

Ezt a megszakító alprogramot intézi, tehát ha ezzel akarunk dolgozni, a megszakítást nem kapsolhatjuk ki. Az

```
IF PEK (653) AND 2 ↑ (bit sz.) THEN PRINT "BILLENTYŰ
                                          LENYOMVA"
```

utasítássorral BASIC-ből megállapítható, hogy valamelyik billentyűt lenyomtuk-e vagy sem.

Összefoglalás: Billentyűk egyidejű lekérdezése

Két lehetőségünk van:

a) PEEK (653) – három gomb egyidejű lekérdezése

b) Az 56320, X utasítással kiválasztjuk az oszlopot (4. ábra szerint), majd az 56321-es rekesz kiolvasásával megállapítjuk, hogy az adott oszlopból melyik billentyűt nyomtuk le. Erre egy egyszerű példa az M 26 (Melléklet).

9.3. A billentyűk kiiktatása

Sokszor jó lenne, ha némelyik billentyűt (legtöbbször a RUN/STOP-ot) vagy az egész billentyűzetet kiiktathatnánk.

Ehhez a 64-es több lehetőséget is kínál.

A teljes billentyűzet hatástalanításához kikapcsolhatjuk a megszakító-rutint. Ilyenkor még a kurzor is eltűnik és a gép látszólag érzéketlen, Ezt az állapotot a RUN/STOP-RESTORE – billentyűk egyidejű lenyomásával szüntethetjük meg.

Ugyanezt elérhetjük a POKE 649,0 utasítással is, de ilyenkor a kurzor továbbra is előállítható (mesterségesen is). A RUN-STOP-gomb is hatásos marad. Érdekes az utasítás működése. A 649-es rekesz határozza meg a billentyűzet-puffer maximális hosszát, ami eredetileg 10 jel. Ha ezt 0-ra csökkentjük, az operációs rendszer úgy tekinti, mintha a puffertároló tele lenne és nem fogad el további bevitelt, azaz a billentyűk lenyomására nem reagál.

Azonos eredményt érünk el a POKE 655, 71 utasítással is. Ezzel ugyanis megváltoztatjuk a dekódertáblázat mutatóját. Következménye: a megszakító-rutin nem tudja többé előállítani az ASCII-kódokat – a billentyűzet-puffer üresen marad. Visszaállítása POKE 655, 72 utasítással végezhető.

Ha csak a RUN/STOP-billentyűt akarjuk kikapcsolni, megtehetjük a POKE 788, 52 utasítással. Ezután a BASIC-program csak a RUN/STOP-RESTORE együttes lenyomásával állítható meg. Feladása: POKE 788, 49.

A POKE 792,193 utasítással egy mini-reset-et (STOP és RESTORE) idézhetünk elő. A STOP-gomb továbbra is hatásos marad.

A két utóbbi utasítás kombinációjának hatására a BASIC programok már egyáltalán nem állíthatók le (kivéve a ki-/bekapcsolást). Ha a programot még listázás ellen is védeni akarjuk, a POKE 808,234 utasítást kell alkalmaznunk.

Ezután az összes megszakítási lehetőség hatástalan lesz és a LIST-utasítás nem ad értelmes eredményt.

Összefoglalás: A billentyűk kiiktatása

A teljes billentyűzet kiiktatása:

1. A megszakítás kikapcsolása
2. POKE 649,0 (billentyűzet-puffer hossza 0)
3. POKE 655,71 (dekódertáblázat-mutató módosítása)

RUN/STOP	kiiktatása	: POKE 788,52
	visszakapcsolása	: POKE 788,49
RESTORE	kiiktatása	: POKE 792,193
	visszakapcsolása	: POKE 792,71
BREAK	kiiktatása+listavédelem	: POKE 808,234

9.4. Az ismétlési funkció (REPEAT)

Ezt akkor alkalmazzuk, amikor a kurzort a képernyő egy távolabbi pontjára visszük. Szeretnénk ezt más billentyűkkel is megvalósítani? Semmi akadálya! Az ismétlési funkció kiterjesztésében a 650-es tárolórekesz az illetékes. Alapesetben minden bitje 0. Ez a megszakító-rutin számára azt jelenti, hogy csak a kurzor- és szóköz-billentyűk ismételhetők. Ha a 6. bitet bekapcsoljuk (POKE 650,64), az ismétlés minden billentyűre nézve megszűnik. POKE 650,128 utasításra viszont ennek éppen az ellenkezője történik. Ezután az ismétlési funkció a normál billentyűkre (A, S, D ...stb.) is kiterjed. Ezek mellett az apróságok mellett még valamit tudnunk kell. Először is, hogyan hat a billentyűk ismétlése a megszakító-rutinra? Mielőtt egy billentyű automatikusan megismétlődne, eltelik egy kis "nekifutási" idő (kb. 0,5 s.). Ez azért van, hogy ha a gépkezelő véletlenül egy kicsit tovább tartotta lenyomva a billentyűt, ne zavarja meg egy felesleges ismétlés.

Ezt az időtartamot a 652-es rekesz állítja elő. Az ott álló számot (általában 16) a megszakító-rutin visszaszámlálja 0-ig. Csak ezután indul az ismétlés. Ekkor hasonló visszaszámlálás jön létre a 651-es rekeszben is. Amikor itt is elérte a 0-át, egy "új" billentyűlenyomás történik, a pufferbe újabb jel kerül és a 651-es regiszterbe új kezdőérték (4) kerül. A POKE 651,255 utasítással az ismétlést kb. újabb 4 másodperccel késleltethetjük.

Összefoglalás: Az ismétlési funkció

POKE 650,128: minden billentyű ismétlődik
POKE 650,64 : ismétlés kikapcsolása
POKE 650,0 : a kurzor- és a szóköz-billentyűk ismétlődnek (eredeti állapot)
POKE 651,255: az ismétlés késleltetése további 4 s-mal.

9.5. A billentyűzet lekérdezése – másképpen

Az előző fejezetből már tudjuk, hogy a megszakítórutin a lenyomott billentyű ASCII-kódját elhelyezi a billentyűzet-pufferben. Ennek a folyamatnak azonban van egy közbülső állomása – a 203-as tárolórekesz. Ez a közbenső tárolója az ún. billentyűzetkódoknak, amiket a dekódertáblázatban mutatóként használunk. A kód csak addig marad (203) utasítással pl. időfüggő beviteleket programozhatunk, ahol az eredmény a billentyű lenyomásának hosszától függ.

Az 1. táblázatban összefoglaltuk a billentyűzetkódokat, amiknek sajnos nem sok közös vonásuk van az ASCII-kódokkal.

A	10	O	38	2	59	@	46	F5	6
B	28	P	41	3	8	*	49	F7	3
C	20	Q	62	4	11	↑	54	STOP	63
D	18	R	17	5	16	:	45	SPC	60
E	14	S	13	6	19	;	50		
F	21	T	22	7	24	=	53		
G	26	U	30	8	27	RET	1		
H	29	V	31	9	32	,	47		
I	33	W	9	←	57	.	44		
J	34	X	23	+	40	/	55		
K	37	Y	25	–	43	⇓	7		
L	42	Z	12	£ CLR	48	⇒	2		
M	36	Ø	35	~~CIR~~	51	F1	4		
N	39	1	56	DEL	Ø	F3	5		

1. táblázat

Egy gombnyomás, amit a PEEK(203)-mal felfedezünk, ezáltal még nincs kitörölve a billentyűzet-pufferből. GET-tel vagy INPUT-tal hozzárendelhetjük egy változóhoz. Ezzel már a tulajdonképpeni adatbevitel előtt felülvizsgálhatjuk. Ezenkívül a billentyűzetkódok közbenső tárolása csak a megszakítás kikapcsolásával akadályozható meg. Így adott esetben elkerülhető a billentyűzet kiiktatása.

A billentyűzet-puffer egyébként is törölhető. A 198-as rekesz adja a már tárolt ASCII-kódok számát. A POKE 198,0 utasítással egy törlést idézünk elő, mivel minden újonnan érkező jel a puferben egy régit felülír. A törlés

után alkalmazott WAIT 198,1 utasítással a programot a következő gombnyomásig megállíthatjuk. Mihelyt egy új jel érkezik, az a pufferszámlálóban (tehát a 198-as tárolórekeszben) feljegyzésre kerül. A WAIT-utasításának itt csupán az a feladata, hogy észlelje a jel beérkezését és ezután szabaddá tegye a programfutást. Ezután a jelet GET-tel beolvashatjuk a pufferből. Ezzel a módszerrel felesleges IF ... THEN szerkezeteket takaríthatunk meg.

Maga a billentyűzet-puffer a 631 ... 640 tárolótartományban található. A jelek itt ASCII-kódként kerülnek rögzítésre, amiket azután a BASIC átvehet. Ha ezeket POKE-kal írjuk be, gombnyomást szimulálhatunk. Ilyenkor azonban a 198-as rekeszben lévő mutatót is megfelelően emelni kell, különben a BASIC nem fogja megtalálni a jelet.

Amint látjuk, a billentyűzet lekérdezése során nagyon sokoldalú. Ismereteink alapján bátran kísérletezzünk!

Összefoglalás: A billentyűzet lekérdezése

PEEK (203)	– megadja az éppen lenyomott billentyű ún. billentyűzet-kódját
POKE 198,0	– a bilentyűzet-puffer törlése
POKE 198,0:WAIT 198,1	– várakozás egy billentyű lenyomására

A billentyűzet-puffer a 631 ... 640 tártartományban helyezkedik el. Ha ide a billentyűk ASCII-kódjait POKE utasítással visszük be, gombnyomást szimulálhatunk.

10. A BOTKORMÁNY (JOYSTICK), PADDLE, FÉNYCERUZA ÉS EGYEBEK

Ezeket mindenki ismeri, de kevesen vannak, akik azt is tudják, hogyan működnek. A játékok és grafikák kiegészítő eszközeiről van szó. Legelterjedtebb közülük a botkormány, ami nélkül – sokak szerint – a C–64-es nem is teljes értékű.

Következzék tehát a leírásuk, amiből mind a működésüket, mind pedig a lekérdezési technikát elsajátíthatjuk.

10.1. A botkormány

Ezen sokan csodálkoznak, de igaz: a botkormány a 64-esnek tulajdonképpen csak egyféle második billentyűzete.

Lekérdezése ugyanis az egyik billentyűzet-oszlopon keresztül történik.

Mindkét botkormány bemenet (-port) a CIA 1-hez csatlakozik.

Az 1. számún az 56321-es regisztert visszajelzésre használjuk. A különböző helyzeteknek (jobbra, balra, fel, le, tűz) a 7. billentyű-oszlop (ld. 4. ábra) felel meg. Amikor tehát az 1-es bemenetet akarjuk lekérdezni, az 56320-as regiszterben 127-nek kell lennie. Ez mindig akkor áll elő, amikor a megszakító-rutin befejezte a billentyűzet lekérdezését. Abban az esetben viszont, ha előzőleg a billentyűzet lekérdezése az 56320/1 rekeszen keresztül történt, a botkormány használata előtt be kell írnunk a

POKE 56320, 127

utasítást.

A botkormány állásától függően törlődnek az 56321-es rekesz bitjei.

A következő táblázat a bitek hozzárendelését mutatja:

bit	7	6	5	4	3	2	1	0
irány	–	–	–	tűz	jobbra	balra	le	fel
billentyű	–	–	–	szóköz	2	CTRL	←	1

A táblázatban azok a billentyűk szerepelnek, amelyekkel a botkormány szimulálható. Ha a RUN/STOP-billentyűt nem kapcsoltuk ki, akkor a 145-ös rekeszt más célokra hasznosíthatjuk. Itt az operációs rendszer előállítja az 56321-es rekesz másolatát, ezért az 1-es botkormány bemenet a

PEEK (145)

utasítással is lekérdezhető.

Valamivel bonyolultabb a 2-es bemenet működése. Ez az 56320-as rekesszel van összefüggésben, ami viszont az oszlop kiválasztásában (ami egy kivitel) illetékes. A botkormány lekérdezése viszont egy külső bevitelt igényel.

A CIA-nak ezt a bemenetét tehát át kell kapcsolnunk, mégpedig a

POKE 56322,224

utasítással. Ez kétféle következménnyel jár. Egyfelől azzal, hogy most az 56320-as rekeszből is, mint az 56321-esből, a botkormány mozgása olvasható ki. Másrészt pedig kiiktatjuk a billentyűzetet, ami a

POKE 56322,255

utasítással vagy a RUN/STOP–RESTORE billentyűkkel oldható fel.

A botkormány működési módja nagyon egyszerű. Öt nyomógombból áll. Egyik a tűzgomb, a másik négy pedig a négy irányt képviseli. A botkormány állásától függően hol az egyik, hol a másik lép működésbe, amit a 64-es az illetékes tárrekeszeiben rögzít.

Természetesen a kereskedelemben kapható botkormányok sem egyformák. Az egyszerűbbek (mint a Commodore VC-1311 is) kontakt-fóliával dolgoznak (ez az egykori ZX–81 felhasználóiban bizonyára kellemetlen emlékeket idéz). Sokkal komolyabbak azok, amelyekben mikrokapcsolókat alkalmaznak. Ezt a működtetéskor halk kattanás jelzi.

Vásárláskor mindenesetre ügyeljünk arra, hogy a botkormány élei legömbölyítettek legyenek, mert különben a játék közben gyorsan felléphet a kifáradási jelenség. Megjegyezzük, hogy az összes ATARI-kompatibilis botkormány a Commodore-hoz is használható.

Összefoglalás: botkormány

1-es botkormány-bemenet lekérdezése: PEEK (56321)
Ekkor az 56320-as tárolórekesznek 127-et kell tartalmaznia.

2-es botkormány-bemenet lekérdezése:

POKE 56322,224 – átkapcsolás bemenetre
PEEK (56320)

Az 1-es bemenet kisegítésképpen PEEK (145)-tel is lekérdezhető.

10.2. A paddle-k

Ezek az eszközök általában forgatható szabályozóként ismeretesek. A botkormányokkal ellentétben, ami csak egy irányt ad meg, az a feladata, hogy a szabályozó állásától függően egy pozíciót vagy egy értéket adjon át a számítógépnek. Ezt egy potenciométerrel valósítja meg. Leegyszerűsítve: ezt minél jobban elfordítjuk valamelyik irányba, annál jobban átfolyhat rajta a gép felől érkező áram, és fordítva. A C–64-es képes arra, hogy az átfolyó áramot az A/D- (analóg/digitális) átalakítón keresztül mérje és a mérési eredményt egy digitális számmá alakítsa. Ez a szám azután egy speciális regiszterből leolvasható. Az A/D-átalakító is és ez a regiszter is a SID részei.

Mivel portonként két paddle csatlakoztatható (igaz, hogy csak egy dugón keresztül), adott a két átalakító és a két regiszter is. Ezek az 54297-es és 54298-as. Mindkét paddle-nak van tűzgombja is. Ezeket a botkormány "jobb" és "bal" irányaihoz hasonlóan az 56321 (1-es kapu) és az 56320 (2-es kapu; ne felejtsünk el átkapcsolni bemenetre!) regiszterekből olvashatjuk le.

A figyelmes Olvasó már megállapíthatta – ez a leírás még nem teljes. Összesen 4 paddle-t csatlakoztathatunk a C–64-esünkhöz, de csak két átalakítónk van. Ezért kell, hogy legyen valamiféle lehetőségünk a két port közti átkapcsolásra. Van is. Ha az 56320-as regiszter 7. bitjét bekapcsoljuk, akkor a mérési eredmények átvétele a 2. porton jön létre. Ez azonban megint csak akkor valósítható meg, ha megszakító-rutin nem tud beleavatkozni. Következésképpen: ki kell kapcsolni!

Összefoglalás: A paddle-k

A paddle-értékek lekérdezése az 54297 és 54298-as regiszterből lehetséges.

A gombnyomás a botkormány "bal" és "jobb" irányának felel meg.

Az AD-átalakítót az 56320-as regiszter 7. bitjének bekapcsolásával váltjuk át a 2-es portra.

10.3. A fényceruza

Most jön a technika egyik csodája – legalábbis a kívülállók számára annak tűnik. Hogy képes egy ilyen szerény külsejű eszköz, mint amilyen ez a fényceruza, pontokat rajzolni a képernyőre? A lényege nem is olyan bonyolult. A pontok tulajdonképpeni elhelyezését egy program végzi. Ez úgy működik, mint a 6. fejezet grafikus rutinjai (P.14–1). A fényceruzának csak egy feladata marad, hogy megadja a pont koordinátáit.

Ezek két regiszterbe kerülnek. De vajon VIC honnan tudja, hogy a fényceruza éppen melyik képernyőpontra mutat?

Ahhoz, hogy erre a kérdésre válaszolhassunk, néhány dolgot tudnunk kell a tv-kép felépítéséről. Azt már biztosan tudjuk, hogy külön sorokból áll. Ezeken sorra egymásután végighalad egy elektronsugár. Egy pont ábrázolása annyit jelent, hogy be kell kapcsolni. Amikor tehát az elektronsugár egy olyan ponton halad keresztül, aminek meg kell jelennie a képernyőn, bekapcsolódik és az üveg speciális bevonatát gerjesztve világító pontot hoz létre. Ahol a pontnak sötétnek kell lennie, a sugár kimarad. A képernyő letapogatása olyan gyors, hogy szemünk azt állóképként érzékeli. Egy kép felépítéséhez a másodperc töredéke elegendő. Ha a fényceruzát a képernyőre helyezzük, akkor az, amint ráesik az elektronsugár, impulzust küld a VIC-nek. Ez pedig megállapítja, hogy a képernyői képnek éppen melyik pontjáról érkezett a jel. Mivel a videojelet a VIC állítja elő, meg tudja állapítani, hogy éppen melyik koordináták következnek. Ezeket az X- és Y-értékeket a 19-es (53267) és 20-as (53267) VIC regiszterekben rögzíti, ahol a program rendelkezésére állnak. BASIC-ből a PEEK utasítással olvashatók ki.

Az így kapott értékek 0 ...255 között lehetnek. Ezeket raszter koordinátáknak hívjuk és nem egyeznek meg a grafikus koordinátákkal. A koordinátákat az egyszerű hármasszabály elvén átszámíthatjuk, majd aktivizálhatjuk a pontot.

Összefoglalás: A fényceruza

A fényceruza-koordinátákat a 19-es (53267) és 20-as (53268) VIC-regiszterek rögzítik. Felhasználásukkal kidolgozhatunk egy pont rajzolására alkalmas grafikus rutint.

10.4. Egyéb tartozékok

Szakirodalomból már bizonyára ismerős az ún. grafikus asztal. Ezen a felhasználó úgy rajzolhat, mint egy darab papíron, de a kép nagyfelbontású grafikaként megjelenik a képernyőn.

Különböző elven működő változatai vannak. A közös bennük az, hogy felismerik, melyik ponton van éppen a ceruza, toll vagy hasonló. A megállapítás eredménye változó erősségű áramként kerül a paddle-bemenetekre. Innen azután már a számítógép gondoskodik a továbbiakról. Megfelelő szoftver nélkül azonban itt sem boldogulunk.

A paddle-bemenetekre még egy különleges botkormány is csatlakoztatható, amit itt mi proporcionálisnak (arányosságon alapuló) fogunk nevezni. Lényege, hogy nem csak mozgásirányt tud megadni, hanem két koordinátából álló pontos helyzetet is.

Ezt egy X/Y – potenciométerrel valósítja meg. Tulajdonképpen két, közös házba épített potméterről van szó, amelyek közül az egyik az X-, másik az Y-irányt szabályozza.

A botkormány mozgatásakor az iránytól függően változik a két potenciométer értéke.

Ilyen módon a képernyő bármely pontja elérhető.

Ez utóbbi két készülék előnye, hogy kész helyzetkoordinátákat adnak át, ezáltal megtakarítható a hagyományos botkormányoknál szükséges hosszadalmas oda-vissza vezérlés.

11. A USER-PORT

A C 64-es a user-port által nagyon sokoldalúvá válik. Sajnos a kézikönyv ennek használatáról és programozásáról egy szót sem ejt. E nagyon elmarasztalható tény miatt itt legalább a programozás alapjaival meg kell ismerkednünk.

11.1. Általában az interface-ekről

Ugyanúgy, mint a billentyűzet és a botkormány a user-port is egy CIA-n, mégpedig a CIA 2-n keresztül üzemel. A CIA-k úgynevezett interface- vagy I/O-elemek. Ezeknek a chipeknek, az a feladatuk, hogy a perifériákról érkező adatokat átvegyék, majd továbbítsák a processzorhoz és fordítva.

Egy ilyen elem általában három részből áll. Egyik egy párhuzamos kapu (port), amit a billentyűzettel kapcsolatban már megismertünk. A másik az időzítőegység, amit már eddig is szorgálmasan használtunk, anélkül, hogy feltűnt volna. A harmadik egy soros kapu (port). Nézzük meg részletesebben!

11.1.1. A soros kapu (port)

Ez a legegyszerűbb, kezdjük tehát ezzel.

Mint tudjuk, a számítógép a byte-okat párhuzamosan dolgozza fel, azaz a 8 bitet mindig egyidejűleg mozgatja, kezeli ...stb. Ezzel szemben egy soros interface a 8 bitet egy vezetéken, egymás után továbbítja. Ez persze lassúbb, mint egy párhuzamos átvitel, előnye viszont az, hogy 8 helyett 1 adatvezeték is elegendő, pl. egy telefonkábel. (Így lehetséges a telefonvezetéket adatátvitelre igénybe venni.)

Az interface-elem feladata a különböző adatformák közti átalakítás. A processzor az illető biteket párhuzamosan adja át az IC-nek, amiket az, megfelelő ütemben, egymás után (sorosan) továbbít.

Fordítva: az IC fogadja az egymás után beérkező biteket, míg össze nem jött egy byte. Ekkor ezt együtt továbbítja a processzornak.

Ha mindezt magának a processzornak kellene végrehajtania, a soros buszon keresztül az adatcsere rendkívül lassúvá válna, mert minden bit átvételéhez több gépi kódú utasítást is végre kellene hajtania.

A soros interface igazán hatékonyan csak gépi nyelven programozható, de mivel az ehhez szükséges elmélettel nem rendelkezünk, erre itt nem térünk ki. Ezenkívül a 64-es ROM-ja egy soros RS 232 típusú interface (dugaszolható modulként kapható) kezelésére alkalmas teljes szoftvert tartalmaz, amivel azt egy egyszerű OPEN 1,2 utasítással aktivizálhatjuk.

11.1.2. Az időzítő

Mindig, amikor valamilyen belső időlefutást kell szabályózni, működésbe lép a számítógép belső időzítő berendezése. Regisztereibe tetszőleges időértékeket tölthetünk, amelyet folyamatosan visszaszámol. Amikor a nullát elérte, egy megfelelő jelet küld a processzornak. A belső időzítés egyik jellemző példája a megszakítás (aha !). Az órát úgy programozták, hogy minden 1/60 másodpercben "ébreszt", majd újra kezdi a visszaszámlálást. Az "ébresztésre" a processzor megszakítja a főprogramot és a megszakítás-alprogramba ugrik – no lám!

Ezzel kapcsolatban térjünk vissza az 1. fejezethez és nézzük meg mégegyszer, hogyan lehetséges a megszakítás kikapcsolása. Az 56334-es tárolórekesz 0. bitje határozza meg, hogy az óra folytassa-e a visszaszámlálást, vagy leálljon. Ha a bitet kikapcsoljuk (és éppen ezt teszi a POKE-utasítás), az időmérés egyszerűen abbamarad. Végeredménye: többé nincs megszakítás.

Ettől a trükktől eltekintve lehetőleg ne nyúljunk az időzítőhöz, mert a legtöbb esetben a gép kezelhetetlenné válik.

11.1.3. A párhuzamos kapu (port)

A 16502 ill. 6510 típ. processzorok interface-einek van egy közös tulajdonsága: a párhuzamos kapuk programozása. Egy IC általában két ilyen párhuzamos kapuval rendelkezik, mint a CIA-k is.

Minden kapu 8 adat-csatornát tartalmaz, amik bemenetként vagy kimenetként programozhatók. Ehhez a chipnek két speciális regisztere van. Az adatirány-regiszter megmutatja, hogy az egyes csatornák melyik üzemmódban működnek. Az 1-es bitek kimenetet, a 0-ás bitek bemenetet jelentenek.

Ennek a hozzárendelésnek különleges oka van. Ha a 0 jelentené a kimenetet, előfordulhatna, hogy a számítógép bekapcsolásakor véletlen impulzu-

sok kerülnek a perifériákra. Ilyenkor ezek működésbe léphetnek és tönkre tehetik a lemezre rögzített adatállományunkat. A második regiszternek az üzemmódtól függően különböző feladatai vannak. Átmeneti tárként szolgál, amelyből a proceszszor bevitelkor átveszi a perifériáról érkező adatokat, illetve ahová a kivitelre szánt információkat betölti. Arra, hogy a processzor közölje az illetékes perifériával, hogy az adatok átvételre készek, az ún. handshake (kézfogás-) eljárás alkalmas. Amikor a processzor az I/O-egységnél leadta az átvitelre szánt byte-ot, egy speciális vezetéken (handshake-vezeték) keresztül közli a perifériával, hogy átveheti. A következő byte-tal ezután mindaddig vár, míg a periféria (szintén egy handshake – vezetéken keresztül) visza nem jelez, hogy az átvételt befejezte. Az eljárás akár egy, akár két vezetéken futhat.

Az I/O-elemek emellett még egyéb funkciókat is ellátnak, pl. impulzusok adását és vételét. Az adatátvitelnek sem kell feltétlenül a handshake-eljárás szerint lezajlania.

11.2. Hogy használjuk a user-portot?

A user-porttal egy párhuzamos kapu és különböző kiegészítő csatornák állnak rendelkezésünkre. E csatornák legtöbbje azonban belső jelek továbbítását végzi, emiatt korlátozzuk úgy hogy egy 8 bites kapu és egy "kölcsönkért" vezérlőcsatorna áll rendelkezésünkre. Ez azért "kölcsönkért", mert a CIA 2 A kapujáról származik, tehát tulajdonképpen egy adatcsatorna.

A CIA 2 báziscíme 56576. Ez egyben az A kapu adatregiszter címe is (0. regiszter), ahol a 2-es bit határozza meg a vezérlőcsatorna állapotát. Ennek a kapunak az összes többi csatornáját a gép belső folyamatokhoz veszi igénybe, ezért csak ezt az egy bitet (2. bit) befolyásolhatjuk.

Más a helyzet az 1. regiszterrel (56577). Ez a B kapu adatregisztere, ami tulajdonképpen az igazi user-port. Itt mind a 8 csatorna szabadon használható.

Ezután következnek 2-es ás 3-as regiszterszámmal az adatirányregiszterek. Az A kapu részére az 56578 (Vigyázat! Csak a 2. bit módosítható) és a B kapu részére az 56579. Ezeket a már ismert módon használhatjuk: a POKE 56579,255 utasítással mind a 8 adatcsatornát kimenetként, az 56579,0 utasítással pedig bemenetként programozzuk.

A vezérlőcsatorna programozása valamivel bonyolultabb. A POKE 56578, PEEK (56578) AND 251 utasítás bemenetet, a POKE 56578, PEEK (56578) OR 4 kimenetet eredményez.

A kapun keresztül kivitelre szánt adatokat egyszerűen be kell írnunk az 56577-es rekeszbe és a kivitelről érkező adatokat is ugyanonnan olvashatjuk ki.

A vezérlőcsatornára érkező áramot a POKE 56576, PEEK (56576) OR 4 utasítással be-, ill. a POKE 56576, PEEK (56576) AND 251 utasítással kikapcsolhatjuk.

Ha a programon belül ezt a két utasítást közvetlenül egymás után alkalmazzuk, egy rövid impulzust állíthatunk elő.

A vezérlőcsatorna kivezetése a user-port M-lába (ld. CBM-kézikönyv), a 8 adatcsatornát pedig a C ... L lábak adják.

Összefoglalás: A user-port programozása

A 8 adatcsatorna irányregisztere : 56579 (B kapu)
A vezérlőcsatorna irányregisztere : 56578 (csak a 2. bit)
A kapu adatregisztere : 56577
A vezérlőcsatorna adatregisztere : 56576 (csak a 2. bit)

11.3. Alkalmazási példák

A user-port nagyon sokoldalúan használható, ezért itt nem kész programokat, hanem csak néhány ötletet adunk.

A legegyszerűbb lámpákból vagy LED-ekből állítható elő, amiket esetleg meghajtó-tranzisztorokon vagy reléken keresztül a user-port-ra csatlakoztatunk. Ezzel pl. fényorgonát készíthetünk, ami az AD-átalakítóra csatolt mikrofonon keresztül érzékeli a zene hangerejét és ennek függvényében ki- ill. bekapcsolja a lámpákat. Egy másik programmal futó fények és egyéb hatások valósíthatók meg.

Elképzelhető két (típustól független) Commodore-gép összekapcsolása, adatcsere céljából. Megvalósítható pl. az, hogy a VC 20-on végzett méréseket a C–64-es nagyobb képernyőjén, nagyfelbontású grafikával ábrázoljuk.

Az elektronika kedvelői készíthetnek maguknak egy saját soros buszt, amivel az adatokat pl. telefonvezetéken továbbíthatják.

Ugyancsak elképzelhető, hogy a C 64-eshez nem Commodore típusú nyomtatókat (pl. telexgépet, lyukszalag lyukasztót és olvasót, személyi-robotot vagy zsebszámológépet) kapcsoljunk.

Egy ezermester fantáziáját semi sem korlátozza!

12. A BASIC ÉS AZ OPERÁCIÓS RENDSZER

A BASIC és az operációs rendszer sok olyan funkcióval áll rendelkezésünkre, amelyek az eddig leírtakkal nincsenek összefüggésben. Gyakran szükséges azonban, hogy bizonyos célokra ezeket is (pl. LIST) befolyásolni tudjuk. A következőkben ezekkel a lehetőségekkel ismerkedünk meg.

12.1. BASIC sorok előállítása programmal

Tételezzük fel, hogy szeretnénk egy olyan programot írni, ami egy tetszőleges függvénygörbét rajzol a nagyfelbontású képernyőre. Ha a programba nem írjuk be eleve adottként a függvényt, akkor meg kell teremtenünk annak a lehetőségét, hogy azt a programfutás közben megadhassuk. Legegyszerűbb lenne, ha a felhasználó írná be egy speciális programsorba a DEFFN segítségével. Ez azonban némi programozási ismeretet feltételez. Kényelmesebb lenne INPUT-tal bekérni, de ez sajnos azért nem jó, mert az így megadott függvény a továbbiakban füzérként szerepel, így viszont nem kivitelezhető. Utolsó lehetőség az lenne, ha a számítógép önmagát programozhatná. Ez egész egyszerűen megvalósítható.

Ahhoz, hogy az eljárást jobban megértsük, nézzük először is, hogyan jön létre egy normál programsor. Az egész azzal kezdődik, hogy a felhasználó begépel egy sor (remélhetőleg) előre átgondolt betűt és jelet. Ezek egyidejűleg a képernyőn is megjelennek. Ha a begépelt jel egy RETURN, a BASIC-értelmező az egész képernyősort (nem csak a bevitt jeleket) átveszi a BASIC – beviteli pufferbe és átalakítja programsorrá, ill. sorszám híján közvetlenül végrehajtandó utasításokká.

Az értelmezőnek mindegy, hogy a képernyőn megjelenő jeleket úgy gépeltük-e be, vagy valamilyen PRINT-utasítás eredményeként kerültek oda. Erre épül az eljárásunk. A képernyőn először is meg kell jelennie a programsor szövegének, amit ezután már csak át kell alakítanunk valódi programsorrá. Ehhez szükségünk van egy mesterségesen előállított gombnyomásra, amit úgy kapunk, hogy a billentyűzet-pufferbe "bePOKE-oljuk" az illető billentyű ASCII-kódját. Ha most a programban egy END-utasítás következik, a program megszakítása után ez a gombnyomás kerül végrehajtásra. Itt két probléma adódik.

Az új programsor létrehozásával, ugyanúgy, mint egy normál programbevitelkor, törlődnek a régi változók. Ez úgy kerülhető el, hogy a programsor mesterséges előállításának műveletét a program elejére építjük be, amikor még nincsenek a tárban fontos adatok. Ha ezt nem tudjuk megoldani, akkor a szükséges adatokat POKE-kal be kell írnunk olyan szabad tárterületekre, amiket az operációs rendszer nem használ.

A másik, hogy a programnak az új programsor előállítása után is tovább kell futnia. Ehhez egy mesterséges GOTO XXX utasításra van szükségünk, amit ugyanúgy állíthatunk elő, mint a példánkban szereplő programsort.

Nézzünk egy erre vonatkozó listát:

```
1 REM *** P 20 ***
2 REM
10 INPUT"KEREM A FGV.-T: Y=";A$
20 PRINT"[illegible]100 DEFFNF(X)=";A$:REM A SOR KIVITELE
30 PRINT "GOTO 70[illegible]";:REM UTASITAS A PR. FOLYT.-HOZ
40 POKE 631,13:POKE 632,13:REM RETURN 2-SZER
50 POKE 198,2:REM BILL.PUFFER INICIALIZALASA
60 END
70 REM ....
READY.
```

Ha begépeltük és kipróbáltuk, pillanatok alatt rájövünk az utasítások céljára, különös tekintettel a képernyő kivitelre. Az így előállított programsor semmiben sem különbözik a hagyományosan előállítottaktól. A program tetszőlegesen sokszor lefuttatható. Hibás programszöveg bevitelére az új programsor feldolgozásakor a gép SYNTAX ERROR hibaüzenettel reagál. Ez a program még jelentősen kibővíthető. Megvalósítható a feleslegessé vált programsorok törlése és egyidejűleg több új programsor is létrehozható. Ezzel az eljárással lehetséges egész alprogramok INPUT-tal való bevitele is.

12.2. Védelem, listázás ellen

Személyes vagy más titkos adatokkal dolgozó programokba ajánlatos jelszót vagy jelkódot beépíteni az illetéktelen hozzáférések elkerülésére. Természetesen ennek csak akkor van értelme, ha ez a kód listázással nem állapítható meg. Ezt a sort tehát listázás ellen védenünk kell, ami ugyancsak egy POKE-utasítással lehetséges.

A megértéshez ismernünk kell a programsor tárón belüli formátumát. A sor első két byte-ja képezi mindig a következő sor mutatóját. Így lépkedhet az értelmező sorról-sorra. Ha mindkét byte 0, nincs több programsor, itt van tehát a program vége.

A mutató után következő két byte tárolja a programsor számát. Ez ugyanúgy épül fel, mint a mutató. Ezután következnek az értelmezőkódokká lefordított utasítások. A sor végét egy 0 jelzi. Ez az, amivel becsaphatjuk az értelmezőt. Ha ugyanis közvetlenül a sorszám után lévő byte-ba POKE-kal egy nullát írunk, akkor a LIST-alprogram ezt úgy tekinti, mint a sor végét és rögtön a következő sorra ugrik (a mutatót, a sor elején álló két byte-ot, nem változtattuk). Ezzel az esetleges GOTO-utasítást sem befolyásoljuk, mivel az az alprogram, amelyik a szövegben egy adott sort keres, szintén a mutatók alapján tájékozódik.

Más a helyzet a program feldolgozásakor. Az a rutin, amelyik a végrehajtás során a következő utasítást keresi, a nulla után egyszerűen átugrik 4 byte-ot. Ezért "hiányzik" programfutáskor a sorok első négy byte-ja. Hogy az utasítások végrehajtását meg ne akadályozzuk, a védendő sor elejére 5 tetszőleges jelet (de ne utasításszót!) kell írnunk. Az elsőt a 0 felülírja, a többinek helyfenntartó szerepe van.

De honnan tudjuk, hogy melyik byte-ot kell felülírnunk? Nos, erre is van egy trükk. Beírunk a védendő sor elé egy STOP-utasítást és elindítjuk a programot. A BREAK után a 61/62-es rekeszekből kiolvasható a következő BASIC-utasítás mutatója. Ha a STOP a sor végén áll, a mutató a sorvégre, azaz egy 0-ra mutat. Adjunk ehhez a címhez ötöt és máris megvan a keresett byte címe.

Tehát munkára fel:

POKE cím, 0

Ezután listázáskor ennek a sornak már csak a száma fog megjelenni, a szövege nem. Ki kell még törölnünk a feleslegessé vált STOP-utasítást és kész is vagyunk.

Összefoglalás:

1. A sor elé egy STOP-ot teszünk.
2. A sor elejére 5 tetszőleges jelet írunk (utasításszó nem lehet!).
3. Cím = PEEK (61) + 256 * PEEK (62) + 5
4. POKE cím, 0
5. STOP utasítást töröljük.

Ha az egész programot védeni akarjuk, a nulláslapon kell módosítanunk a LIST-alprogram mutatóját. Így a gép ezt nem fogja megtalálni, következésképp a programot képtelen lesz listázni. A mutató a 774/775-ös rekeszekben van. Ezt a POKE 775,1 utasítással úgy eltorzíthatjuk, hogy minden listázási kísérlet RUN/STOP–RESTORE-ként fog hatni. POKE 775, 167 utasítással állíthatjuk vissza az eredeti állapotot.

12.3. RENUMBER

Egyes BASIC-bővítések, mint az EXBASIC, vagy a SIMON'S BASIC, rendelkeznek az igen jól használható utasítással: a RENUMBER-rel. Ezzel átsorszámozható a tárban levő program, aminek pl. a MERGE-utasítás használatakor van nagy jelentősége.

A RENUMBER azonban BASIC-bővítő hiányában is, legalább utánozható. Mint az előző fejezetben már említettük, a tárban minden programsor két mutatóval kezdődik. Az első a következő sor elejére mutat, a második pedig, ami valójában nem is mutató, az adott sor száma mutató-formájában megadva. Adjunk 2-t a következő sor mutatójához és megkapjuk a következő sor kezdőcímét. Ilyen módon az összes programsort végignézhetjük és POKE-kal tetszés szerint módosíthatjuk. Az ehhez szükséges program a következő:

```
1 REM *** P 21 ***
2 REM
63900 BA=PEEK(43)+256*PEEK(44)
63910 INPUT"KEZDO SSZ.";KS:INPUT"LEPTEK";LP
63920 HI=INT(KS/256):LO=KS-HI*256
63930 A=PEEK(BA+2)+256*PEEK(BA+3)
```

```
63940 IF A>=63900 THEN PRINT"RENDBEN!":END
63950 POKE BA+2,LO:POKE BA+3,HI
63960 BA=PEEK(BA)+256*PEEK(BA+1):KS=KS+LP
63970 PRINT KS"="A:GOTO 63920
READY.
```

Ezt az alprogramot MERGE-utasítással vagy begépeléssel hozzáfűzzük az átsorszámozandó programhoz és RUN 63900-zal indítjuk. A sorszámozást azért választottuk ilyen magasra, hogy mindig a főprogram végére kerüljön.

A 63900-as sor a BASIC-kezdőcím mutatójából kiszámítja az első sor báziscímét. A 63910-es sor bekéri a felhasználótól az átsorszámozás kezdő értékét és a léptéket. Ha azt akarjuk, hogy a sorszámozás 10-zel kezdődjön és tízesével emelkedjen, kétszer 10-et kell beadnunk. A 63920-as sor kiszámítja az új sorszám alsó és felső byte-ját, a 63930-as pedig kiolvassa a tárból a régit. Ha ez nagyobb, vagy egyenlő 63900-zal, a program megszakad, hogy az alprogram önmagát ne tudja átsorszámozni.

A 63950-es sor beírja a megfelelő rekeszbe az új sorszám alsó és felső byte-ját.

Végül kiszámítja a következő sor báziscímét, a léptékkel megemeli a sorszámot (63960. sor) és kiírja az átsorszámozási listát (63970. sor). Ez utóbbi azt mutatja meg, hogy az új sorszámnak mi a régi megfelelője. Ennek használátával az új sorszámozáshoz igazítjuk a GOTO és GOSUB ugrócímeket, mert ez az alprogram ezeket nem tudja módosítani.

Nyomtatóval is rendelkezők mindezt papírra is vihetik, ha az alprogram elejére beiktatják az OPEN 1,4 : CMD 1 utasítássort.

(A könyvben szereplő többi alprogramhoz hasonlóan az 1 . . . 2 sorok csak a tájékozódást segítik. Ezeket a program használatakor ne gépeljük be! – a magyar változat készítői.)

12.4. RENEW

A számítógép tulajdonosok egyik réme a NEW-utasítás. A számítógépeknek van egy közös vonásuk: ennek a három betűnek (N–E–W) a meggondolatlan bevitele (+ RETURN), tönkreteheti az egész eddigi munkát, ha a bevitt programot előzőleg elfelejtettük tárolni. Szerencsére van egy kis programunk, amivel a NEW-katasztrófa visszafordítható, az így törölt program feléleszthető!

A programozók bizonyára tudják, hogy a NEW-utasítás nem jelenti a tár teljes kiürítését, csakis a két fő mutató visszaállítását. Az első ezek közül a változók elejére mutat. NEW után ez visszaáll a program elejére, ami azt eredményezi, hogy az ezután használt változók felülírják a programot, ami egyelőre még mindig a tárban van.

Ezért legelső és legfőbb törvény: a tévesen bevitt NEW után semmi olyat (sem utasítást, sem más jelet) nem szabad bevinnünk, aminek nincs köze a RENEW-hoz! Már egy betű + RETURN is változót képez, annak ellenére, hogy a gép SYNTAX ERROR-t jelez!

A második mutató a program első sorában áll, pontosabban a 2049/2050 rekeszekben. Normál esetben a következő programsorra mutat, most viszont két 0-t tartalmaz, ami a program végét jelzi. A RENEW-hoz tehát két teendőnk marad:

1. Megkeressük az első sor végét, amit egy 0 jelez. Ha megtaláltuk, akkor ennek a címéhez hozzá kell adnunk 1-et mutatóként be kell írnunk a 2049/2050 rekeszekbe.

2. Megkeressük a program végét. Ezt arról ismerjük fel, hogy a következő sor mutatójának felső byte-ja 0. Ha megtaláltuk, a címét 2-vel megnöveljük és így megkapjuk a változótartomány kezdőcímét. Ezután betölthetjük a helyes mutatót.

Nézzük a programot:

```
1 REM *** P 22 ***
2 REM
10 AD=2052
20 AD=AD+1:IF PEEK(AD)<>0 THEN 20
30 AD=AD+1
40 POKE2049,AD-INT(AD/256)*256:POKE2050,INT(AD/256)
```

```
50 IFPEEK(AD+1)<>0THENAD=PEEK(AD)+256*PEEK(AD+1):
   GOTO50
55 C=AD+2
60 PRINT"P┌45,"C-INT(C/256)*256":P┌46,"INT(C/256)":
   P┌44,8:P┌56,160:CLR"
70 PRINT"□□□□"
READY.
```

Néhány megjegyzés a listához. A 20. sor keresi meg az első sor végét. Ha megtalálta, a 30. sorban ennek címéhez 1-et ad és a 40. sorban helyreállítja a második sor mutatóját. Az 50. sor mutatóról mutatóra halad, míg az 0 nem lesz. Itt a program vége. Az így megtalált címet nem írhatjuk be közvetlenül POKE-kal, mert akkor az alprogram önmagát tenné működésképtelenné, hanem az utasításokat képernyőre visszük (60. sor), majd a kurzorral átmegyünk rajta (70. sor). Már csak a RETURN-t kell lenyomnunk és a változó-mutató ismét a megfelelő helyre mutat.

Van azonban még egy probléma. Ha ezt így, ahogy van, begépeljük, tönkre tesszük az éppen felújításra váró programunkat. Először tehát át kell helyeznünk a BASIC-tárát egy olyan területre, amit az értelmező nem használ. Erre a célra éppen megfelel a 49152 ...53247 közti 4 k RAM. Ahhoz, hogy a BASIC-területet mindennel együtt oda áthelyezhessük, az alábbi 4 utasításra van szükségünk:

POKE 44,192: POKE 56,208: POKE 49152, 0 : NEW

Ezzel teremtettünk magunknak egy második, független tár-tartományt és begépelhetjük a P 22-es programot.

Mielőtt azonban elindítanánk, tároljuk. Így a későbbiekben elegendő lesz csak betölteni az áthelyezett BASIC-területre. Betöltése: LOAD "P 22",8,1.

Most már indíthatjuk (RUN). Ha a BASIC-kezdőcímet már eleve eltoltuk (pl. HI–RES grafika részére), akkor csupán a 10. és 20. sorban levő báziscímeket és a 60. sorban levő POKE-utasítást kell módosítanunk.

A POKE-utasítások vissza is helyezik a BASIC-tárat (POKE 44,8).

Ezután a régi program rendesen használható.

Véletlen NEW (+ RETURN) után tehát következő teendőink vannak:

1. POKE 44,192 : POKE 56,208 : POKE 49152,0 : NEW : REM BASIC-tár áthelyezése.
2. LOAD "P 22",8,1
3. RUN
4. RETURN
5. A régi program újra használható.

A P 22 program 49152-től továbbra is a tárban marad, amiről a mutatók átállítása után (1. pont NEW nélkül) LIST-tel meggyőződhetünk, de mégegyszer nem futtatható. Ha újra használni akarjuk, az egész eljárást (1 ... 4) meg kell ismételnünk.

Ott kell azt is megjegyeznünk, hogy ha a C–64-est a külső RESET-gombbal egy összeomlásból "felélesztettük", a RETURN akkor is használható és így az elveszett program visszakapható. Amíg az áramellátást ki nem kapcsoljuk, minden adat rendelkezésünkre áll.

12.5. RESTORE

Ezt az utasítást csak ritkán használjuk. Néha előfordul, hogy a DATA-mutatót vissza kell állítanunk, de ilyenkor is célszerűbb lenne, ha a sorszámot, vagy magát a beolvasandó DATA-elemet adhatnánk meg, azaz ha a mutatót nem a legelső, esetleg már szükségtelen adatokra kellene visszaállítanunk. Így elkerülhetnénk, hogy a kívánt adat eléréséhez az előtte álló DATA-elemeket is be kelljen olvasnunk.

Néhány POKE-utasítással ezt is megoldhatjuk. Ehhez tudnunk kell, hogy az értelmező mind az aktuális DATA-sor számát, mind pedig a következő DATA-elem címét a nulláslapon tárolja. A sorszámot, a mutatóhoz hasonlóan, a 63/64-es, a következő DATA-elem mutatóját pedig a 65/66-os byte-okban.

Ha most egy RESTORE-hoz hasonló dolgot akarunk végrehajtani, a következőket kell tennünk:

1. Beolvastatjuk a kívánt elem előtti adatokat (pl. közvetlen üzemmódban). Ha az 5. elemre vagyunk kíváncsiak, akkor négyet

2. PRINT PEEK (63), PEEK (64)
   Ezek a számok adják a sorszámot, jegyezzük meg őket!

3. PRINT PEEK (65), PEEK (66)
   Ezeket is meg kell jegyezni! Ezek adják a következőként beolvasandó DATA-elem mutatóját. Az eddigi utasításokat mind a tulajdonképpeni programfutást megelőzően kell bevinnünk.

4. POKE 63,1. szám: POKE 64,2. szám

5. POKE 65,3. szám: POKE 66,4. szám

Ezeket az utasításokat beépítjük a programba a RESTORE helyére.

Általuk a DATA-mutató a beolvasásra szánt elemre áll. A BASIC ezt úgy veszi, mintha az ezt követő DATA-sorokat még nem olvasta volna.

Sajnos az eljárásnak van egy hátránya. Amennyiben az adott DATA-sor előtt bármilyen változtatást végeztünk a programsorokban, módosulni fognak a címek, amiket a DATA-mutatóban meg kell adnunk, mert ilyenkor a tárban a teljes programszöveg eltolódik. Célszerű tehát a DATA-blokkot a program legelején elhelyezni.

Összefoglalás: RESTORE

Az aktuális DATA-sor számát a 63/64, a következő DATA-elem címét pedig a 65/66 rekeszek rögzítik. Mindkét mutató POKE-kal módosítható, miután a kívánt mutatóértékeket megállapítottuk. (l. Melléklet: M27)

12.6. Néhány trükk

Programmegszakítás után vagy hibaüzenet során a gép jelzi, hogy melyik sorban lépett ki a programból. Ha meggondolatlanul töröljük a képernyőt, akkor ez a sorszám általában elveszik. Amennyiben mégis szükségünk lenne rá, megtudhatjuk az 59/60 tárrekeszek kiolvasásával. Ezek rögzítik (mutatóformátumban) az utolsóként végrehajtott sor számát. Kiolvasása a szokott módon végezhető.

```
PRINT PEEK (59) + 256 * PEEK (60)
```

A következő utasítássorral megakadályozhatjuk a program tárolását:

```
POKE 801,0: POKE 802,0: POKE 818,165
```

Ezáltal úgy eltorzítjuk a SAVE-hez szükséges vektorokat, hogy minden tárolási kísérlet meghiúsul.

Hátránya: egy egyszerű RUN/STOP–RESTORE bevitel is kezelhetetlenné teszi a gépet.

Végül néhány jól használható SYS-utasítás:

SYS 65 499: Nullázza a TI$ óraváltozót. Sokkal gyorsabb, mint a "000000" füzér hozzárendelése.

SYS 42 115: A programot END helyett ezzel is befejezhetjük. "Melegindítás"-t eredményez, ami nem más, mint átkapcsolás közvetlen üzemmódra. Nem jelenik meg a READY és a kurzor a következő sorban van. A CONT-utasítás hatástalan marad.

SYS 58 253: Ezzel az utasítással is befejezhetjük a programot. Ekkor megjelenik a bekapcsoláskor megszokott kép és egyúttal törlődik a program.

SYS 44 808: Mesterséges SYNTAX ERROR.

Összefoglalás: Trükkök

Az utolsó végrehajtott sor számát az 59/60-as rekeszek tárolják.

SAVE – védelem: POKE 801,0 : POKE 802,0 : POKE 818,165
TI$ nullázása: SYS 65 499
Programvég READY nélkül: SYS 42115
SYNTAX ERROR : SYS 44808

12.7. BASIC-bővítők

A BASIC-bővítőkről már minden Commodore-tulajdonos hallott, ha nem is rendelkezik ilyesmivel. Ezek közül az elsők már a PET (vagy CBM 2000) idejében megjelentek. Eleinte csak úgynevezett toolkit-utasításokat tartalmaztak, amik a program bevitelét (editálását) segítették. Ilyen például az AUTO, amelyik a megadott léptékkel automatikusan képezi és kiírja a programsorszámokat, megtakarítva ezzel a begépelési munkát. A FIND kikeres a programszövegben egy meghatározott kifejezést. Ilyen még a MERGE, RENUMBER és RENEW, amelyek feladatával már az előző fejezetekben megismerkedtünk. A DEL-lel teljes programrészeket törölhetünk, a TRACE-szel követhetjük a programfutást, a DUMP listázza a változókat és pillanatnyi értéküket.

A komolyabb változatok segítik a hibakezelést, lehetővé teszik a hibás bevitel korrigálását anélkül, hogy az hibaüzenethez és programmegszakításhoz vezetne.

Ritkábban azzal a lehetőséggel is találkozunk, hogy a funkcióbillentyűkhöz tetszőleges jelsorozatokat rendelhetünk.

Mivel a 64-es BASIC-je a gép fantasztikus hang és grafikai lehetőségeit nem támogatja kellően, sok BASIC-bővítés nyújt ehhez segítséget.

Néhány bővítő a strukturált programozáshoz ad utasítástokat, amelyekkel GOTO nélkül programok írhatók. Ebben az esetben egy-egy programrészt ún. modulként szerkesztünk meg, hasonlóan az alprogramokhoz. Egy pont megrajzolását végző rutinra ekkor a GOSUB helyett pl. a CALL PLOT X, Y utasítással hivatkozhatunk. Ezzel a technikával jól áttekinthető (strukturált) programokat készíthetünk.

Elterjedtek a speciális DOS-utasításokat kínáló bővítők is. Ezek közt van, amelyik pl. a lemez tartalomjegyzékét közvetlenül a képernyőre viszi, anélkül, hogy a tárban lévő programot törölné ... stb.

12.8. Egyéb programnyelvek

A 64-es egyik jellemzője, hogy BASIC-től eltérő nyelven írt programokat is képes betölteni és feldolgozni. Ezek közül legismertebb a PASCAL. Alapelve a strukturált programozás, ami azt jelenti, hogy igyekszik kiküszöbölni a GOTO-utasítást (egyes PASCAL-változatok ezt egyszerűen nem is tartalmazzák). Legfőbb érdeme, hogy a program modulokból (előregyártott programrészekből) építhető fel. Ez azért is jó, mert megakadályozza a programozót abban, hogy előzetes elgondolás nélkül hozzáfogjon a program írásához.

A PASCAL mindig compiler-ként jelenik meg, ami azt jelenti, hogy lefuttatás előtt a gép a programszöveget lefordítja valamilyen gép-orientált nyelvre. Ez legtöbbször gépi nyelv, vagy egy gyors közbülső nyelv.

Ezzel szemben a FORTH kimondottan értelmező-nyelv. Itt is nagy súlyt kell fektetnünk a strukturált programozásra, mivel néhány saját utasítást is definiálnunk kell (de nem gépi nyelven). A FORTH mindössze néhány alaputasításból áll, emiatt nagyon közel áll a gépi nyelvhez, és rendkívül gyors.

A LOGO szintén nagyon elterjedt. Olyan egyszerűen megtanulható, hogy akár elsősosztályos gyerekek is boldogulnak vele. Fő jellemzői: A Turtlegrafika (a képernyőn egy képzeletbeli teknősbékát mozgatunk, úgy mint egy ceruzát és ezzel egyszerűen grafikákat programozhatunk) és a modulrendszer. A LOGO különösen matematikai és geometriai feladatok megoldására alkalmas.

13. A GÉPI NYELV

Aki a C–64-es programozásával komolyabban akar foglalkozni, nem lehet meg a gépi nyelv ismerete nélkül. Sok kezdő számára azonban nehéz a gépi kódban való gondolkodás. Ezzel a fejezettel nekik szeretnénk segíteni. A könyv végén található szimulátor-programmal kipróbálhatunk egyfajta alap- vagy minimál- nyelvet, ami alapján eldönthetjük, hogy akarunk-e vele komolyabban foglalkozni vagy maradunk a BASIC-nél (ez sem feltétlenül rossz!). Mivel magát a szimulátor-programot BASIC-ben írtuk, természetesen arra nem alkalmas, hogy a gépi nyelv gyorsaságát is szemléltesse. Ehhez inkább a 6. fejezetben található grafika-törlő-alprogramot (P 13) vegyük elő.

13.1. Mi is az a gépi nyelv?

Azt bizonyára mindenki tudja, hogy ez az egyetlen lehetőség arra, hogy a processzort compiler vagy értelmező nélkül; közvetlenül programozzuk. Ezzel érhetünk el rendkívül nagy futási sebességet. A gépi kódú nyelv különböző alapműveletekből áll, amelyekből a magasabb szintű programnyelvek, mint pl. a BASIC, komplex utasításai összeállíthatók. A gépi kódú utasítások durván három csoportba sorolhatók. BASIC-programozók számára legkönnyebben érthetők az ugró-utasítások, amelyek a GOTO-hoz és GOSUB-hoz hasonlóan működtetik a programot. A következő csoportba azok az utasítások tartoznak, amelyek az adatok feldolgozását (összeadás, összekapcsolás ...stb.) végzik. A harmadikat azok a műveletek alkotják, amelyek az adatokat a tárban egyik helyről a másikra mozgatják.

Alapvetően érvényes, hogy a mikroprocesszor számára nem léteznek változók. Csak normál tárrekeszeket és belső regisztereket ismer. Adatfeldolgozás általában csak az utóbbiakban lehetséges.

Egy gépi nyelvű utasítás mindig egy egy byte-os kódszámból (operációs kódnak is nevezzük) áll, amit az operandus részére 1 vagy 2 byte követhet. Egy tárrekesz tehát – a program eddigi futásától függően – utasítást, címet vagy adatot tárolhat.

13.2. Az ütem

A számítógépben mindent egy kis jelentéktelen külsejű kvarckristály irányít, ami az ütemet adja. (0.98 MHz – 980000 ütem vagy ciklus másodpercenként.)

A processzor ütemenként 1 alapműveletet tud elvégezni. Alapműveleteknek nevezzük az egyes gépi nyelvű utasítások hatására lezajló folyamatokat. A legrövidebbek ezek közül 2 ütem hosszúságúak: az "utasítást a tárból elővenni és dekódolni", valamint az "utasítást végrehajtani". Az összetettebb műveletek utasításai több ütemet igényelnek.

13.3. A tizenhatos számrendszer

Bármikor foglalkozzunk is a gépi nyelvvel, mindig találkozni fogunk a számok 16-os számrendszerbeli ábrázolásával.

Ez, mint a nevéből is kitűnik, 16 "számjeggyel" (0–9 és A–F) rendelkezik. Főleg azért használható nagyon jól, mert a bináris- és hexadecimális számok között egyszerű az átszámítás.

A 8 számjegyből álló (egy byte-os) bináris számot két fél byte-ra osztjuk és ezeket egyenként egy hexadecimális számmá alakítjuk. A következő táblázatban felsoroljuk az első 16 szám bináris, decimális és hexadecimális megfelelőjét:

bináris	decimális	hexadecimális
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Az 10101011 értékű byte-ból az AB hexadecimális szám lesz.

A hexadecimális szám decimálissá való átalakításakor először minden számjegyet külön-külön decimálissá alakítunk, majd mindegyiket megszorozzuk a helyiértékével (16 hatványai), végül az egészet összeadjuk.

Nézzünk egy példát:

Alakítsuk át az ABCD hexa-számot decimálissá!

$$\begin{aligned} ABCD &= 10*16\uparrow 3 + 11*16\uparrow 2 + 12*16\uparrow 1 + 13*16\uparrow 0 = \\ &= 10*4096 + 11*256 + 12*16 + 13*1 = 43\ 981 \end{aligned}$$

Ez fordítva is elvégezhető. Ebben az esetben a decimális számot folyamatosan addig osztjuk 16-tal, míg 0-t nem kapunk. Az osztások maradékait jegyezzük hexa-számjegyekként, fordított sorrendben.

Példa:

5300:16=	3312	maradék	8———8
3312:16=	207	maradék	0———0
207:16=	12	maradék	15———F
12:16=	0	maradék	12———C

Tehát $53000_{dec} = CFO8_{hex}$

A kereskedelemben már sok olyan zsebszámológép létezik, amelyik rendelkezik a bázisátlalakítás speciális funkciójával. A jobb assembler- illetve monitor-programok is képesek rá.

13.4. Bináris összeadás

Mindjárt az elején le kell szögeznünk: A bináris összeadás csupán a számrendszerben tér el a decimálistól, maga a folyamat ugyanúgy zajlik le.

Két nulla vagy egy nulla és egy egyes (a sorrend lényegtelen) összege egyértelmű, a szokásos módon adjuk össze őket. Amikor viszont két egyest akarunk összeadni, akadályba ütközünk. A tízes számrendszerben az eredmény 2 lenne, de ilyen számjegy a kettes számrendszerben nincs. Itt tehát egy átvitel jön létre, ugyanúgy, mint amikor a tízes számrendszerben az eredmény 9-nél nagyobb. Tehát:

$$\begin{array}{r} 1 \\ +1 \\ \hline 10 \end{array}$$

Ez teljes byte-ok összeadásánál is ugyanilyen egyszerű:

$$\begin{array}{rr} 0110\ 1110= & 110 \\ +0000\ 1001= & 9 \\ \hline 0111\ 0111= & 119 \end{array}$$

Előfordulhat, hogy két 1-est kell összeadnunk és ehhez még egy átvitel is társul (azaz 1+1+1). Az eredmény 3, azaz 11 (világos!). Próbáljuk ki:

$$\begin{array}{r} 1001\ 0011 \\ +1101\ 1111 \\ \hline 10111\ 0010 \end{array}$$

Nahát, most meg egyszerre 9 bitünk lett! Ezt a kilencedik bitet nevezzük átviteli (vagy Carry-) bitnek. Ez megmutatja, hogy két 8-bites szám összeadásának eredménye meghaladta-e egy byte megengedet értéktartományát (0–255). Ezt adott esetben hozzá kell adnunk egy másik byte-hoz. Ezzel el is érkeztünk a 16-bites számok összeadásához. A számítógépek 8-bittel nem tudnak megfelelően nagy számtartományt ábrázolni. Tény azonban, hogy a 8-bites mikroprocesszorok (ilyen a 6510-es is) egyszerre mindig csak 8 bitet képesek feldolgozni. Ha egy szám két byte-ból áll, akkor ezek összeadását egymás után kell elvégezni. Mivel a szám két részét egymásután, egymástól teljesen függetlenül adhatjuk össze, a carry-bitet csak nagyobb számok feldolgozásánál használjuk. Feladata az, hogy az első byte utolsó helyéről a második byte első helyére kerülő átvitelt ideiglenesen tárolja.

Példa:

$$\begin{array}{rr} 00110101 & 10010011 \\ +10011011 & 11011111 \\ & \\ 1111111 & 11111 \quad \text{(átvitelek)} \\ \hline 11010001 & 01110010 \end{array}$$

Segítségképpen feltüntettük az átviteleket is. Az összeadás jobboldali részét az előző példából már ismerjük.

13.5. Bináris kivonás

Amikor a számítógépnek egy számból egy másikat ki kell vonnia, akkor először képezi az utóbbi negatív megfelelőjét és ezt hozzáadja az előbbihez. Ez azért van így, mert az elektronikus alapelemekből (mint AND, OR, XOR, NOT) összeadás és tagadás (negálás) előállítható, de kivonás nem. A negatív számok ábrázolásához egy byte számtartománya 0 ... 255 helyett − 128 ... +127.

A legmagasabb helyiértékű bit (a 7-es) ekkor előjelként szolgál. Ha 1, negatív, ha 0, pozitív számról van szó.

Egy szám negálásához azonban nem elegendő, hogy a 7-es bitjét 1-re változtassuk. Nézzük meg ezt egy példán keresztül.

$$\begin{array}{r} 00000001 \\ +10000001 \\ \hline 10000010 \end{array}$$

Decimális rendszerbe átültetve ez annyit jelentene, hogy 1−1=−2. Emiatt egy másik utat kell választanunk.

Nagyon egyszerűen megvalósítható egy szám −1-gyel való szorzása. Ez úgy történik, hogy minden bitet invertálunk és ráadásként egy 1-est hozzáadunk.

Példa:

$$\begin{array}{rr} & 01011011 \\ \text{invertálva:} & 10100100 \\ + & 1 \\ \hline & 10100101 \end{array}$$

Végezzük most el ennek alapján az 1−1 kivonást bináris alakban:

$$\begin{array}{r} 00000001 \\ +11111111 \\ \hline 100000000 \end{array}$$

Most helyes eredményt kaptunk, de sajnos egy átvitel is keletkezett. A kivonás azonban ebből a szempontból is más. Ha a carry-bit értéke 1, akkor ez azt jelenti, hogy nincs átvitel, a 0 viszont tartomány túllépést jelent. Emiatt a carry-bitet minden kivonás előtt 1-re kell állítani.

Kivonási utasítás végrehajtásakor a 6510-es mikroprocesszor a carry-bit beállításán kívül minden egyéb feladatot automatikusan elvégez.

13.6. Magasabbrendű matematikai műveletek

Alig hihető, de igaz: A 6510-es gépi nyelve csak két számítási műveletet ismer, az összeadást és a kivonást. Minden mást erre a két alapműveletre vezet vissza. Az X*n szorzatot például úgy számítja ki, hogy X-et n-szer összeadja. Ez persze csak egész számok esetén lehetséges. Tört számok esetén komplikáltabb algoritmus szükséges (a számok helyiértékük szerint és nem mint egészek lesznek összekapcsolva), de az elv hasonló.

Ha X-et osztani akarjuk n-nel, akkor a gép folyamatosan kivonja X-ből n-et, míg egy számláló jegyzi a kivonások számát. A végrehajtható kivonások száma adja az osztás eredményét.

Példa:

10/3=?

10–3=7	számláló	regiszter	1
7–3=4	számláló	regiszter	2
4–3=1	számláló	regiszter	3

⇒10/3=3, maradék 1.

Ez a módszer a gépi nyelv rendkívüli gyorsasága miatt elfogadható. A zsebszámológépek is ezen az elven működnek. Amikor egy billentyűt lenyomunk, mindig lefut egy kis gépi nyelvű program a szükséges algoritmussal együtt.

A négy alapműveletből magasabbrendű műveleteket (pl. hatványozás, színusz stb.) is összeállíthatunk. Ilyen módon AND-del, OR-ral, XOR-ral és NOT-tal mindenféle matematikai művelet kifejezhető.

13.7. Összehasonlítások

BASIC-ben nem szokatlanok az összehasonlítások. De hogy van ez gépi kódban? Nézzünk erre egy példát:

$A = B \Rightarrow A - B = 0$

Mint látjuk, két szám (itt A és B) összehasonlítása nagyon egyszerűen átalakítható. A gép számára az utóbbi forma előnye az, hogy az egyenlet jobb oldalán 0 áll. A nulla az egyetlen szám, amiről a mikroprocesszor meg tudja állapítani, hogy éppen ez van-e az akkuban vagy sem. Ez úgy történik, hogy a bitek között sorban egymás után OR-kapcsolatot létesít, valahogy így:

7. bit OR 6. bit OR 5. bit OR 4. bit OR 3. bit OR 2. bit OR 1. bit OR 0. bit

Ha az akku mind a 8 bitje 0, az összekapcsolás eredménye is 0, ellenkező esetben (azaz, ha bármelyik bit 1) az eredmény 1.

Ebből a mikroprocesszor meg tudja állapítani, hogy az akku tartalma (ami csaknem mindig az utolsó művelet eredménye) egyenlő-e 0-val, vagy attól eltérő. Ahhoz tehát, hogy egy A = B egyenlőséget vagy egy A <>(nem egyenlő) B egyenlőtlenséget megállapíthassunk, a két számot kivonjuk egymásból, majd megvizsgáljuk, hogy az akku tartalma egyenlő-e 0-val. A "nagyobb" és "kisebb" összehasonlítás hasonlóképpen történik. A kivonás után azt kell megjegyeznünk, hogy az akku tartalma kisebb-e vagy nagyobb, mint nulla.

Ezt az előjelbitről állapíthatjuk meg:

A nagyobb, mint B $\Rightarrow$ A–B nagyobb, mint 0 (a 7. bit=0)
A kisebb, mint B $\Rightarrow$ A–B kisebb, mint 0 (a 7. bit=1)

13.8. A szimulátor utasításai

Miután megteremtettük az elméleti alapokat a gépi nyelvű programozáshoz, meg kell ismerkednünk azokkal az utasításokkal, amelyekkel a szimulátor-programon (P 23) belül dolgozhatunk. Alkalmazzuk ezeket bátran, nem baj, ha valamelyiket nem értjük meg rögtön az első nekifutásra.

Az utasításokat most is három csoportba soroljuk. Mindegyikhez rövid magyarázatokat fűzünk, közöljük a helyes és lehetséges írásmódokat, ezenkívűl az utasításszó után megadjuk a rövid angol jelentését is.

13.8.1. Az adatkezelés utasításai

ADC: Add With Carry

Ez az utasítás az utána következő operandust és az átviteli bitet az akku tartalmához adja. Az átvitelt az átviteli bit rögzíti.

Az operandus kétféle lehet. Vagy a hozzáadandó bit címét adjuk meg, vagy a szám a tárban közvetlenül az utasítás után áll.

. Lehetséges alakjai:
ADC $HH (a HH tárrekesz tartalmának hozzáadása az akku tartalmához)
ADC #HH (a HH szám hozzáadása az akku tartalmához)

SBC: Subtract With Carry

Ugyanúgy működik, mint az ADC, de kivonást végez.
Lehetséges alakjai:
SBC $HH (a HH tárolórekesz tartalmának kivonása az akku tartalmából)
SBC #HH (a HH szám kivonása az akku tartalmából)

AND: And With Accu

Az operandus és az akku tartalma között ÉS-kapcsolatot létesít. Az átviteli bitet nem veszi figyelembe. Az operandusra ugyanaz vonatkozik, mint az ADC-nél.

Lehetséges alakjai:
AND $HH
AND #HH

ORA: Or With Accu

Ugyanaz, mint az AND, de VAGY-kapcsolattal.
Lehetséges alakjai:
OR $HH
OR #HH

EOR: Exclusive OR with accu

Ugyanaz, mint az AND, de kizáró-, VAGY-kapcsolattal.
A bináris 1111 1111 értékkel NOT-ként működik.
Lehetséges alakjai:
EOR $HH
EOR #HH

DEC: DECrement

A megadott című byte-ot 1-gyel csökkenti. Ha az eredmény 0, beállítja a nulla-bitet, ill. negatív számok esetén a negatív-bitet. Az átviteli bit nem változik.

Lehetséges alakja: DEC $HH.

DEX: DEcrement X

Ugyanaz, mint a DEC, de az X-regisztert csökkenti.
Lehetséges alakja: DEX.

INC: INCrement

Ugyanaz, mint a DEC, de növelést (+1) jelent.
Lehetséges alakja: INC $HH.

INX: INcrement X

Ugyanaz, mint az INC, de az X-regisztert növeli.
Lehetséges alakja: INX.

CLC: Clear Carry

At átviteli bit törlése.
Lehetséges alakja: CLC.

SEC: SET carry

Az átviteli bitet 1-re állítja.
Lehetséges alakja: SEC.

ASL: Arithmetic Shift Left

Az akku minden bitjét 1 hellyel balra tolja. A 7. bit az étviteli bitbe kerül, a 0. bit értéke 1.
Lehetséges alakja: ASL.

LSR: Logical Shift Right

Az akku minden bitjét 1 hellyel jobbra tolja. A 0. bit az átviteli bitbe kerül, a 7. bit értéke 0.
Lehetséges alakja: LSR.

13.8.2. Ugró utasítások

JMP: JuMP

A program a megadott címen (HH) folytatódik.
Lehetséges alakja: JMP $HH.

JSR: Jump to SubRoutine

Ugrás a megadott címen kezdődő alprogramba.
Lehetséges alakja: JSR $HH.

RTS: ReTurn from Subroutine

Visszatérés az alprogramból (esetünkben a szimulátor programba).
Lehetséges alakja: RTS.

BCC: Branch on Carry Clear

Programelágazás a megadott címre, ha az átviteli bit 0.
Lehetséges alakja: BCC $HH.

BCS: Branch on Carry Set

Programelágazás a megadott címre, ha az átviteli bit 1.
Lehetséges alakja: BCS $HH.

BEQ: Branch on EQual to zero

Programelágazás a megadott címre, ha a nulla-bit értéke 1.
A nulla-bit azt adja meg, hogy az utolsó művelet eredménye egyenlő 0-val. Ha igen, a nulla bit értéke 1, ha nem, akkor 0.
Lehetséges alakja: BEQ $HH.

BNE: Branch on Not Equal to zero

Programelágazás a megadott címre, ha a nulla-bit értéke 0.
Lehetséges alakja: BNE $HH.

BMI: Branch on MInus

Programelágazás a megadott címre, ha a negatív-bit értéke 1.
A negatív-bit azt mutatja meg, hogy az utolsó művelet eredménye kisebb-e, mint 0. Ha igen, a negatív-bit értéke 1, ha nem, akkor 0.
Lehetséges alakja: BMI $HH.

BPL: Branch on PLus.

Programelágazás a megadott címre, ha a negatív-bit értéke 0.
Lehetséges alakja: BPL $HH.

13.8.3. Az adatmozgatás utasításai

LDA: LoaD Accu

Az argumentumot az akkuba tölti. Az argumentum lehet cím, vagy egy közvetlen érték. Az előző esetben az akkuba az adott című byte értéke, az utóbbiban a közvetlenül megadott érték kerül.

Lehetséges alakjai:
LDA $HH (a HH című byte értékét az akkuba tölti)
LDA #HH (a HH értékét az akkuba tölti)

LDX: LoaD X

Ugyanaz, mint az LDA, de az X-regiszterre vonatkozóan.
Lehetséges alakjai:
LDX $HH (a HH című byte értékét az X-regiszterbe tölti)
LDX #HH (a HH értéket az X-regiszterbe tölti)

STA: STore Accu

Az akku tartalmát a megadott című byte-ba tölti.
Lehetséges alakja: STA $HH.

STX: STore X

Ugyanaz mint STA, de az X-regiszterre vonatkozóan.
Lehetséges alakja: STX $HH.

TAX: Transfer Accu into X

Az akku tartalmát az X-regiszterbe tölti.
Lehetséges alakja: TAX.

TXA: Transfer X into Accu

Az X-regiszter tartalmát az akkuba tölti.
Lehetséges alakja: TXA.

13.9. A szimulátor-program

A következő oldalakon megtaláljuk a gépi nyelv-szimulátor program listáját. Ennek segítségével megismerkedhetünk egy kicsit a gépi kódú programozással. Maga a program BASIC-ben íródott, ezért meglehetősen lassú. Esetleg valamilyen compiler-rel gyorsabbá tehető.

Indítás után a szimulátor assembler-üzemmódban van, beadhatók a gépi nyelvű utasítások. Ehhez a képernyő felső harmadában levő beviteli sorba a következőket kell beírnunk

Cím utasítás operandus

Az egybyte-os utasításoknál elmarad az operandus. Az egyes bevitelek között 1–1 üres helyet kell hagynunk. A bevitelt RETURN-nel zárjuk. Hibás bevitel esetén a beviteli sor jobb szélén három kérdőjel jelenik meg. Egy tetszőleges billentyű lenyomására a kérdőjelek eltűnnek és írható a következő utasítás.

A képernyő felső harmadában a fontosabb regiszterek és bitek jelennek meg. A címtartomány utolsó négy byte-ja hexadecimális, bináris és ASCII-kód formájában kerül kijelzésre.

Ez alatt láthatók a programszámláló (PC) az akku (AC) az X-regiszter és a különböző bitek (átviteli-, negatív-, nullat-bit). Az utolsó kijelzés a TRACE-állapotra vonatkozik (be/ki). Minden szám hexadecimálisként értendő!

Nem gépi nyelvű utasítás bevitele előtt le kell nyomni a "←"-billentyűt. Ezután következhet az utasítás ismertetőjele (egy betű) és esetenként az operandus.

Itt elsőként a C-utasítást kell megemlítenünk. Ezzel törölhetjük a képernyő alsó felét. Ügyeljünk arra, hogy az utasítások bevitele során olyan billentyűket, mint pl. a CLR, ne használjunk, mert ezzel tönkretehetjük a képernyő-maszkot.

A következő utasítás a T. Ezzel a TRACE-üzemmódot kapcsolhatjuk ki és be.

A G cím-utasítással indítjuk el az adott címen található gépi nyelvű programot.

A D cím-utasítással disassemblálhatjuk ill. kilistáztathatjuk a két cím közti programrészt.

A Z cím byte-utasítással az adott címhez hozzárendelhetjük az adott byte-ot.

13.10. Az első program

Gépi nyelvvel való ismerkedésünket az összeadással és kivonással kezdjük, mert ezek a legalkalmasabbak arra, hogy a processzor működését megvilágítsák. Nézzünk tehát egy olyan programot, ami két számot ad össze.

A két összeadandót előzőleg az FF és a FE byte-okban rögzítjük. Ez közvetlen üzemmódban, a Z-utasítással történik. Nem a legkényelmesebb megoldás, de a célnak megfelel.

Gépeljük be tehát sorban az alábbiakat:

← Z ␣ FF (vagy FE) ␣ szám (hexadecimálisan)
RETURN.

Amennyiben a sor jobb szélén megjelenik a három kérdőjel, valamit rosszul csináltunk. Ez akkor is előfordulhat, ha a bevitel során a kurzorral egy másik sorba mentünk. Ha mindent jól csináltunk, az utasítás végrehajtása után az érték megjelenik a regiszterben, amit a képernyőn is láthatunk. Mindkét számot ugyanígy kell beírni.

Feltűnhet, hogy a beviteli sor elején mindig megjelenik egy hexa-cím. Ez az a cím, amit a gép a következőként beadandó assembler-utasítás tárolási címeként javasol. Mielőtt az utasítást bevisszük, meg kell gondolnunk, hogy hogyan is nézzen ki.

Az assembler-programozás fő alapelve, hogy (szinte) minden adatkezelési művelet az akkuban történik. Ezért a program első utasításának az akkuba kell töltenie az egyik összeadandót.

Ez az LDA $FF utasítás végzi. Előfordulhat, hogy az átviteli bit értéke még 1, ezért azt a CLC-utasítással törölnünk kell. Ezután következhet a tulajdonképpeni összeadási utasítás, az ADC $FE. Ez az utasítás kiolvassa az FE tárrekesz tartalmát és hozzáadja az akkuban lévő számhoz. Az eredmény ezután ismét az akkuba kerül. Mivel az eredményt a képernyőn is látni szeretnénk, az akkuból betöltjük az FD rekeszbe: STA $FD, aminek tartalma szintén megjelenik a képernyő felső harmadában állandóan látható 4 byte valamelyikében. Ezzel az összeadást elvégeztük, már csak vissza kell térnünk assemblerbe: RTS.

A képernyőn az átviteli bit értékének változását is követhetjük.

A teljes program a következő:

```
LDA $FF
CLC
ADC $FE
STA $FD
RTS
```

Rögzítsük a fenti utasításokat a 00-tárolócímtől kezdődően. Amikor ez a cím a beviteli sor elején megjelenik, hagyjunk ki egy üres helyet, majd gépeljük be az utasítást és a hozzátartozó operandust, pontosan úgy, ahogy azt a lista mutatja.

A bevitelt RETURN-nel zárjuk.

Ha a sor elején más cím áll, egyszerűen írjuk felül a 00-val, majd folytassuk a munkát az előzőek szerint. Ezzel a módszerrel most már az egész program begépelhető és bármikor disassemblálható.

A disassemblálást a ←D kezdőcím végcím-utasítássorral hajthatjuk végre. Ha az így visszakapott listában hibát fedezünk fel, a kérdéses utasítást adjuk be újból.

A program ezek után G 00-utasítással indítható, de előtte ne felejtsük el lenyomni a "←"-billentyűt. A program lefutása után a regiszterekben az aktuális érték található (amit a képernyőn is figyelemmel kísérhetünk) és ismét megjelenik a kurzor.

A programfutást lépésről-lépésre nyomon követhetjük, ha bekapcsoljuk a TRACE-üzemmódot. Ekkor a képernyő jobb alsó sarkában megjelenik az éppen feldolgozott utasítás és a szimulátor egy billentyű lenyomására vár. A képernyő felső harmadában megfigyelhetjük, hogy az utasítás végrehajtása milyen változást idézett elő a regiszterekben és bitekben.

13.11. A második lépés: a 16-bites összeadás

Mint már említettük, nagyobb számoknál a 16-bites összeadást, vagy valamilyen bonyolult algoritmust kell használnunk. A fejezet végén bemutatunk egy olyan programot, amellyel egy tetszőleges 16-bites számot egy állandóhoz adhatunk. A számot a Z-utasítással ismét az FF (felső byte) és az FE (alsó byte) rekeszbe töltjük. Mivel a szám 16 bites, két byte-ra és két összeadási műveletre van szükségünk. Az első lépések azonban ugyanazok:

```
LDA $FE
CLC
```

Mivel egy állandó érték hozzáadásáról van szó (ami szintén lehet 16 bites és így alsó és felső byte-ból állhat), most az ADC # alsóbyte utasítás következik. Ez az utasítás nem egy meghatározott tárrekeszből veszi az értéket, hanem közvetlenül az utasításszót követő számmal dolgozik. Végrehajtása után megvan az eredmény első fele, az alsó byte. STA $FC-vel ezt az FC tárolórekeszbe töltjük, ami a képernyőn is megjelenik.

Átvitel is keletkezhet, amit az átviteli bit jelez. Az LDA $FF utasítás az átviteli bitet nem törli. Ezután következik a második lépés, a belső byte-ok összeadása.

Ez ugyanúgy történik, mint egy normál összeadás, tehát: ADC # felső byte. Megvan tehát az eredmény másik fele is, amit STA $FD-vel az FD rekeszbe töltve a képernyőről leolvashatunk. RTS-sel a program befejeződik.

A teljes lista a következő:

```
00 LDA $FE
02 CLC
03 ADC #E8
05 STA $FC
07 LDA $FF
09 ADC #03
0B STA $FD
0D RTS
```

Állandóként 03E8-at ($= 1000_{dec}$) választottunk.

13.12. Kivonás

Mivel a kivonás az átviteli bit és az utasításszó kivételével ugyanaz, mint az összeadás, itt csak egy rövid programot közlünk. (8- bites művelet)

```
00  LDA  $FF
02  SEC        (átviteli bit bekapcsolása)
03  SBC  $FE   (kivonás)
05  STA  $FD
07  RTS
```

13.13. Szorzás

Mint a 13.6. fejezetből már tudjuk, a szorzás többszörös összeadásra vezethető vissza. Ezt kihasználva a szorzási művelet elvégzése mellett a gépi nyelvű alprogramok összeállítására és alkalmazására is példát mutatunk.

A szorzás lényegét tekintve az a lehetőségünk kínálkozik, hogy az összeadást egy önálló alprogramként írjuk meg. Természetesen ez nem feltétlenül szükséges, de célszerű. A többszörös összeadásra ciklust szervezhetünk, amelyben az összeadó-alprogramot annyiszor hívhatjuk le, ahányszor csak kell. Kezdjük az összeadással. Két 8-bites szám összeszorzása 16-bites eredményt ad. Az összeadó-alprogramnak ezek szerint egy egybyte-os számot egy kétbyte-oshoz kell adnia. Leegyszerűsítve úgy képzelhetjük el, hogy a 8-bites szám elé, felső byte-ként, egy 0-t gondolunk és az összeadást úgy végezzük el, mintha mindkét szám 16-bites lenne. Ezzel biztosítjuk az eredmény felső byte-ja felé az átvitelt.

Nézzük a listát:

```
LDA $FE    (az eredmény alsó byte-ja)
CLC
ADC $FC    (a 8-bites szám hozzáadása)
STA $FE    (visszatöltés)
LDA $FF    (az eredmény felső byte-ja)
ADC #00    (0 hozzáadása az átvitel miatt)
STA $FF    (visszatöltés)
RTS        (alprogram vége)
```

Az összeadás végrehajtása után az eredmény az FE/FFrekeszekben van. A 8-bites számot már előzőleg az FC-ben rögzítettük. Most már csak a ciklus hiányzik. Hoszát a szorzó határozza meg, amit a Z-utasítással FD-ben kell rögzítenünk.

Változó hoszúságú ciklus szervezésének legegyszerűbb módszere, ha minden átfutáskor 1-gyel csökkentjük egy speciális regiszter tartalmát. Amikor a visszaszámlálás elérte a 0-t, a program kilép a ciklusból. Erre a célra a X-regiszter a legalkalmasabb. A ciklus elején betöltjük ide az FD rekesz tartalmát (szorzó): LDX $FD.

Ezután következik az alprogram lehívása JSR $ cím-utasítással, ahol címként az összeadó-alprogram kezdőcímét kell megadnunk. Az alprogram lefutása után 1-gyel csökkentenünk kell a számláló (X-regiszter) tartalmát. Ezt egy egybyte-os utasítással tesszük: DEX. Ennek az utasításnak az az

érdekessége, hogy a negatív- és nulla-biteket is módosítja, jelezve, hogy az X-regiszter tartalma 0 vagy negatív. Ezek a vezérlőbitek tehát nem csak az akkura vonatkoznak.

A 2. táblázatban összefoglaltuk, hogy melyik utasítás melyik vezérlőbiteket módosíthatja. A vezérlőbitek segítségével hajthatók végre az ún. feltételes ugrások. Ilyen feltételt a mi programunk is tartalmaz, amikor a ciklus végét X=0-hoz kötjük. Ez azt jelenti, hogy a ciklusutasítások végrehajtását addig ismétli, míg X egyenlő nem lesz 0-val. Ha a nulla-bit értéke 1, ez azt jelenti, hogy az utolsó művelet eredménye 0.

Z=0 esetén ez nullától különböző.

A BNE – utasítás felülvizsgálja a nulla-bitet. Ha ez 0, a program a megadott címre ugrik, ellenkező esetben a következő utasítás végrehajtásával folytatódik.

TRACE-üzemmódban a programszámlálóval (PC) mindezt végig is követhetjük. Ez a regiszter tárolja a következő végrehajtandó utasítás címét.

Az eredeti 6502/6510-assemblerrel szemben az ugró utasításokat itt némileg leegyszerűsítettük, mert normál esetben nem az ugró-címeket adtuk meg, hanem csak a következő utasítás távolságát (pl.: 3-at előre vagy 20-at hátra ...).

Mielőtt a ciklus elkezdődik, törölnünk kell a két eredménybyte-ot. Nézzük a teljes listát:

```
00 LDA #00
02 STA $FF    (FF törlése)
04 STA $FE    (FE törlése)
06 LDX $FD    (FD → X)
08 JSR $OE    (ugrás az alprogramba)
0A DEX        (X - 1)
0B BNE $08    (programelágazás)
0D RTS        (főprogram vége)
0E LDA $FE    (összeadó alprogram)
10 CLC
11 ADC $FC
13 STA $FE
15 LDA $FF
17 ADC #00
19 STA $FF
1B RTS        (alprogram vége)
```

Parancs	C	N	Z
ADC	X	X	X
AND		X	X
ASL	X	X	X
CLC	X		
DEC		X	X
DEX		X	X
EOR		X	X
INC		X	X
INX		X	X

Parancs	C	N	Z
LDA		X	X
LDX		X	X
LSR	X	X	X
ORA		X	X
SBC	X	X	X
SEC	X		
TAX		X	X
TXA		X	X

2. táblázat

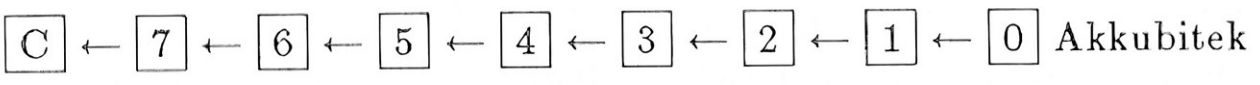
13.14. További lehetőségek

Most már ismerjük az assembler programozás alapelveit. Felesleges hangsúlyozni, hogy az "eredeti" gépi nyelv még jóval több lehetőséget kínál. Lehetséges például az, hogy különböző címzési módokkal a tároló egyes részeit, mint egydimenziós tömböket programozzuk, vagy speciális utasításokkal egyedi megszakító-rutinokat szerkeszthetünk és még sok egyéb.

Ebben a fejezetben ezért csak néhány olyan alapeljárást ismertetünk, amikkel a programozás során gyakran találkozunk majd.

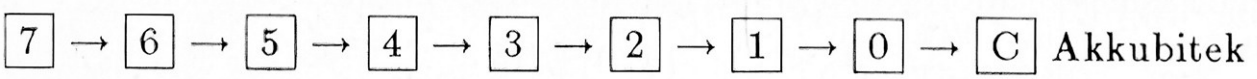
Kezdjük mindjárt az ASL ás LSR utasításokkal. Ezek az ún. toló-utasítások. Feladatuk, hogy az akku bitjeit 1 hellyel balra ill. jobbra tolják. Ezzel az akku tartalma egyszerű módon megduplázható ill. felezhető. Képzeljük el ezt 10-es számrendszerben. Ha a számjegyeket 1 hellyel balra toljuk, az egész szám értéke megtízszereződik. Ugyanez a helyzet a bitekkel is, de mivel itt a kettes számrendszert használjuk, a balra tolás a szám megkétszerezését, a jobbra tolás pedig a felezését jelenti. Az eljárás szabályait az 5. és 6. ábra mutatja.

ASL a 0 bit 0-val tárolódik



5. ábra

LSR 7. bit 0-val tárolódik



6. ábra

A következő utasításokkal már BASIC-ban is találkoztunk. Az AND, OR és EOR utasításokkal, az ADC-hez hasonlóan, összekapcsolhatjuk a tárolóból vett adatokat az akku tartalmával. Így az egyes bitek ki- és bekapcsolásának technikája gépi kódban is alkalmazható.

Az INX-utasítás a DEX rokona, de nem csökkenti, hanem növeli 1-gyel az X-regiszter tartalmát. Ezt inkrementálásnak is nevezik. Itt kell megmagyaráznunk az INC és DEC utasításokat, mivel ezek ugyanúgy működnek, de az X-regiszter helyett egy közvetlenül megadott tárrekeszre vonatkozóan. Ezeknek az utasításoknak az a közös jellemzője, hogy az utolsó bit átvitelét az átviteli bit nem jegyzi.

Amikor egy byte az inkrementálás során elérte az FF értéket, akkor a következő INC-utasítás minden további nélkül 00-tól folytatja a számlálást. Emiatt ezek az utasítások aritmetikai műveletekben alig alkalmazhatók.

A TXA és TAX utasítások hasonlóak. Egyenlővé teszik az akku és az X-regiszter tartalmát úgy, hogy a TAX az X-regiszterbe tölti az akku értékét, a TXA pedig az akkuba tölti az X-et.

Ezek után javasoljuk a további gépi nyelvű kísérletezést. A szimulátorprogram legfőbb előnye, hogy semmilyen körülmények között nem válik kezelhetetlenné (ellentétben az igazi gépi nyelvvel) és a gépi kódú programok RUN/STOP-pal megállíthatók.

13.15. Hogy működnek a SYS-bővítések?

Ezek után, mintegy ráadásként, megbeszélhetjük, hogy programozhatók az olyan utasításbővítések, mint a SYS cím adat.

Egy SYS-utasítás után a belső BASIC-mutató (a programszámlálóhoz hasonlóan) a cím utáni byte-re áll. Egy normál SYS utasítás után az értelmező a gépi-rutinból a visszatérés után folytatná a szintaktika ellenőrzését. Egy JSR $ cím utasítással azonban rávehetjük az értelmezőt, hogy a következő adatokat a legközelebbi vesszőig, kettőspontig vagy a sor végéig beolvassa az ún. lebegőpontos akkuba (ez nem a processzor regisztere, hanem a nulláslap néhány fenntartott rekesze.) A BASIC-mutató most az adatok

utáni byte-on áll. Hibás adat előfordulását a ROM-rutin hibaüzenettel jelzi. Normál számértékek esetében ennek semmi akadálya, az értelmező-ROM ehhez különböző alprogramokkal áll rendelkezésünkre. Egész számú adatok, vagy címek esetében azonban egy további ROM-alprogramra van szükség, ami ezeket a számokat a lebegőpontos akkunak megfelelő formában átalakítja. Ezután a megfelelő időpontban már ezeket is kiolvashatjuk és feldolgozhatjuk a saját gépi nyelvű programunkkal.

Az utasítás-bővítések, mint pl. a SIMON'S BASIC, vagy az EXBASIC ezt a módszert még tökéletesítették. Egy rutin felismeri, kódolja és dekódolja az utasításokat és ezután a megfelelő alprogramba ugrik.

FÜGGELÉK

Gépi nyelvű gyakorló-program

```
1 REM *** P 23 ***
2 REM
1000 REM ********************************
1001 REM *  GEPI NYELV - SZIMULATOR  *
1002 REM *      COPYRIGHT 1984 BY      *
1003 REM *    HANS JOACHIM LIESERT     *
1004 REM *     DATA BECKER TERMEK      *
1005 REM ********************************
1006 :
1010 REM ***************
1011 REM KEPERNYO MASZK
1012 REM ***************
1020 PRINTCHR$(142);CHR$(8):REMPOKE788,52
1030 PRINT"■┌──────────────────────────────────────────────┐";
1040 PRINT "| REG      HEX           BIN         ASC    |";
1050 PRINT "|FF/FE=                                      |";
1060 PRINT "|FD/FC=                                      |";
1070 PRINT "├──────────────────────────────────────────────┤";
1080 PRINT "|PC=   AC=   XR=   C=  N=  Z=  T:            |";
1090 PRINT "└──────────────────────────────────────────────┘";
1100 PRINT "──────────────────────────────────────────────"
1110 PRINT "──────────────┬──────────┬──────────────────";
1120 PRINT " CIM  MNE      |KODOK    |MESSAGES"
1130 FORI=1TO12:PRINT"                |          |":NEXT
1140 PRINT "              |          |";
1150 DIMB(255),C(255),H$(255),EB$(34),EB(34),P1$(10),P2$(10),P3$(10)
1160 FORI=0TO255:READH$(I):B(I)=255:C(I)=255:NEXTI
```

```
1170 FORI=0TO34:READEB$(I),EB(I):NEXTI
1180 FORI=0TO10:P1$(I)="                    ":P2$(I)="      ":NEXTI
1997 REM *********
1998 REM ASSEMBLER
1999 REM *********
2000 GOSUB4000
2010 POKE214,8:POKE211,0:SYS58732
2015 IFAD=256THENAD=0
2020 PRINT"■■■";H$(AD);"■■■■";
2030 INPUT"■■■■■■";IN$
2040 IFLEFT$(IN$,1)="←"THEN2200
2050 PO=1:GOSUB4400:IFEFTHEN4500
2060 AD=0:A$=MID$(IN$,4,3)+"   ":FL=-1
2070 FORI=0TO8:IFA$=EB$(I)THENFL=I
2080 NEXTI:IFFL<>-1THENB(AD)=EB(FL):C(AD)=FL:AD=AD+1:GOTO2000
2090 A$=MID$(IN$,4,5)
2100 FORI=9TO34:IFA$=EB$(I)THENFL=I
2110 NEXTI:IFFL=-1ORAD=255THEN4500
2120 B(AD)=EB(FL):C(AD)=FL
2130 PO=9:GOSUB4400:IFEFTHEN4500
2140 AD=AD+1:B(AD)=0:C(AD)=0
2150 AD=AD+1:GOTO 2000
2197 REM ********************
2198 REM KOZVETLEN UTASITASOK
2199 REM ********************
2200 A$=MID$(IN$,2,1)
2210 IF A$="T" THEN T=1-T:GOTO 2000
2220 IF A$="Z" THEN 2300
```

```
2230 IF A$="D" THEN 2400
2240 IF A$="G" THEN 2500
2250 IF A$="C" THEN 4600
2260 GOTO 4500
2297 REM ****************
2298 REM HOZZARENDELESEK
2299 REM ****************
2300 PO=4:GOSUB 4400:IF EF THEN 4500
2310 AD=0:PO=7:GOSUB 4400:IF EF THEN 4500
2320 B(AD)=0:FL=-1
2330 FOR I=0 TO 34:IF 0=EB(I) THEN FL=I
2340 NEXT I:IF FL<>-1 THEN C(AD)=FL:GOTO 2000
2350 C(AD)=0:GOTO 2000
2397 REM ************
2398 REM DISASSEMBLER
2399 REM ************
2400 PO=4:GOSUB 4400:IF EF THEN 4500
2410 AD=0:PO=7:GOSUB 4400: IF EF THEN 4500
2420 IF AD>0 THEN 4500
2430 H1$=H$(AD):H3$="   ":H4$=H$(B(AD))
2440 IF C(AD)>34 THEN H2$="???  ":GOTO 2470
2450 H2$=EB$(C(AD))
2460 IF C(AD)>8 THEN AD=AD+1:H3$=H$(B(AD))
2470 AD=AD+1:GOSUB 4300
2480 IF AD<=0 THEN 2430
2490 GOTO 2000
2497 REM ************
```

```
2498 REM PROGRAMFUTAS
2499 REM ************
2500 PO=4:GOSUB 4400:IF EF THEN 4500
2510 PC=0
2520 CO=C(PC):AD=PC:PC=PC+1:IF CO>34 THEN 3300
2530 IF CO<9 THEN ON CO+1 GOTO 2700,2720,2730,2770,2790,2810,2830,2840,2850
2540 CO=CO-8:IF CO<3 THEN ON CO GOTO 2860,2880,2900,2910,2920,2940,2960,2980
2550 CO=CO-8:IF CO<9 THEN ON CO GOTO 3000,3020,3040,3080,3100,3120,3140,3150
2560 CO=CO-8:IF CO<9 THEN ON CO GOTO 3150,3170,3180,3190,3200,3210,3220,3240
2570 ON CO-8 GOTO 3260,3270
2590 N=0:IF A>127 THEN N=1
2600 Z=0:IF A=0 THEN Z=1
2610 IF PEEK(203)=63 THEN 3400
2615 IF T=0 THEN 2520
2620 GOSUB 4000:FOR I=0 TO 9:P3$(I)=P3$(I+1):NEXT I
2630 P3$(10)=H$(AD)+" "+EB$(C(AD))
2640 IF C(AD)>8 THEN P3$(10)=P3$(10)+H$(C(AD+1)):GOTO 2660
2650 P3$(10)=P3$(10)+"   "
2660 POKE214,12:POKE211,0:SYS 58732
2670 FOR I=0 TO 10:PRINTSPC(27);P3$(I):NEXT I
2680 POKE 198,0:WAIT 198,1:GET IN$:GOTO 2520
2696 REM **********************
2697 REM ASSEMBLER UTASITASOK
2698 REM **********************
2699 REM ASL
2700 A=A*2:C=0:IF A>255 THEN A=A-256:C=1
2710 GOTO 2590
2719 REM CLC
```

```
2720 C=0:GOTO 2610
2729 REM DEX
2730 X=X-1:IF X<0 THEN X=X+256
2740 Z=0:IF X=0 THEN Z=1
2750 N=0:IF X>127 THEN N=1
2760 GOTO 2610
2769 REM INX
2770 X=X+1:IF X>255 THEN X=X-256
2780 GOTO 2740
2789 REM LSR
2790 A=A/2:C=0:IF INT(A)<>0 THEN A=INT(A):C=1
2800 GOTO 2590
2809 REM RTS
2810 IF S=0 THEN 2000
2820 PC=S(S):S=S-1:GOTO 2610
2829 REM SEC
2830 C=1:GOTO 2610
2839 REM TAX
2840 X=A:GOTO 2740
2849 REM TXA
2850 A=X:GOTO 2590
2859 REM ADC#
2860 A=A+B(PC)+C:C=0:IF A>255 THEN A=A-256:C=1
2870 PC=PC+1:GOTO 2590
2879 REM ADC$
2880 A=A+B(C(PC))+C:C=0:IF A>255 THEN A=A-256:C=1
2890 PC=PC+1:GOTO 2590
2899 REM AND#
```

```
2300 A=A AND B(PC):PC=PC+1:GOTO 2590
2309 REM AND $
2310 A=A AND B(C(PC)):PC=PC+1:GOTO 2590
2319 REM BCC $
2320 IF C=0 THEN PC=B(PC):GOTO 2610
2330 PC=PC+1:GOTO 2610
2339 REM BCS$
2340 IF C=1 THEN PC=B(PC):GOTO 2610
2350 PC=PC+1:GOTO 2610
2359 REM BEQ$
2360 IF Z=1 THEN PC=B(PC):GOTO 2610
2370 PC=PC+1:GOTO 2610
2379 REM BMI$
2380 IF N=1 THEN PC=B(PC):GOTO 2610
2390 PC=PC+1:GOTO 2610
2399 REM BNE$
3000 IF Z=0 THEN PC=B(PC):GOTO 2610
3010 PC=PC+1:GOTO 2610
3019 REM BPL$
3020 IF N=0 THEN PC=B(PC):GOTO 2610
3030 PC=PC+1:GOTO 2610
3039 REM DEC$
3040 H=B(C(PC)):H=H-1:IF H<0 THEN H=H+256
3050 Z=0:IF H=0 THEN Z=1
3060 N=0:IF H>127 THEN N=1
3070 B(C(PC))=H:C(C(PC))=H:PC=PC+1:GOTO 2610
3079 REM EOR#
3080 H=A OR B(PC):A=A AND B(PC)
```

```
3030 A=NOT(A):A=H AND A:PC=PC+1:GOTO 2590
3100 H=A OR B(C(PC)):A=A AND B(PC)
3110 GOTO 3030
3119 REM INC$
3120 H=B(C(PC)):H=H+1:IF H>255 THEN H=H-256
3130 GOTO 3050
3139 REM JMP$
3140 PC=B(PC):GOTO 2610
3149 REM JSR$
3150 S=S+1:S(S)=PC+1:PC=B(PC):GOTO 2610
3159 REM LDA#
3160 A=B(PC):PC=PC+1:GOTO 2590
3169 REM LDA$
3170 A=B(C(PC)):PC=PC+1:GOTO 2590
3179 REM LDX#
3180 X=B(PC):PC=PC+1:GOTO 2740
3189 REM LDX$
3190 X=B(C(PC)):PC=PC+1:GOTO 2740
3199 REM ORA#
3200 A=A OR B(PC):PC=PC+1:GOTO 2590
3209 REM ORA$
3210 A=A OR B(C(PC)):PC=PC+1:GOTO 2590
3219 REM SBC#
3220 A=A-B(PC)-1+C:C=1:IF A<0 THEN A=A+256:C=0
3230 PC=PC+1:GOTO 2530
3239 REM SBC$
3240 A=A-B(C(PC))-1+C:C=1:IF A<0 THEN A=A+256:C=0
3250 PC=PC+1:GOTO 2590
```

```
3259 REM STA$
3260 B(C(PC))=A:C(C(PC))=A:PC=PC+1:GOTO 2610
3269 REM STX$
3270 B(C(PC))=X:C(C(PC))=X:PC=PC+1:GOTO 2610
3296 REM ************
3297 REM RUN-TIME-HIBA
3298 REM ************
3300 POKE214,8:POKE211,0:SYS58732:PRINT"BAD CODE ERROR IN";H$(PC-1)
3310 POKE198,0:WAIT198,1:GET IN$:GOTO 2000
3396 REM *****
3397 REM BREAK
3398 REM *****
3400 POKE214,8:POKE211,2:SYS58732:PRINT"BREAK IN";H$(PC);"       ";
3410 POKE198,0:WAIT198,1:GET IN$:GOTO 2000
3997 REM ********************
3998 REM REGISZTEREK KIJELZESE
3999 REM ********************
4000 H$=H$(B(255))+" "+H$(B(254))
4010 POKE214,2:POKE211,9:SYS58732:PRINTH$;"  ";
4020 H$="":FOR J=255 TO 254 STEP-1:H=B(J)
4030 FOR I=7 TO 0 STEP-1:IF(2↑I AND H)THEN H$=H$+"1":GOTO 4050
4040 H$=H$+"0"
4050 NEXTI:H$=H$+" ":NEXTJ
4060 PRINTH$;" ";
4070 H1$=CHR$(B(255)):IF B(255)<32 OR B(255)>127 AND B(255)<160 THEN H1$=" "
4080 H2$=CHR$(B(254)):IF B(254)<32 OR B(254)>127 AND B(254)<160 THEN H2$=" "
4090 PRINTH1$;" ";H2$
```

```
4100 H$=H$(B(253))+" "+H$(B(252))
4110 POKE214,3:POKE211,9:SYS58732:PRINTH$;"   ";
4120 H$="":FOR J=253 TO 252 STEP-1:H=B(J)
4130 FOR I=7 TO 0 STEP-1:IF(2↑IANDH) THEN H$=H$+"1":GOTO 4150
4140 H$=H$+"0"
4150 NEXTI:H$=H$+" ":NEXTJ
4160 PRINTH$;" ";
4170 H1$=CHR$(B(253)):IF B(253)<32 OR B(253)>127 AND B(253)<160 THEN H1$=" "
4180 H2$=CHR$(B(252)):IF B(252)<32 OR B(252)>127 AND B(252)<160 THEN H2$=" "
4190 PRINTH1$;" ";H2$
4200 POKE214,5:POKE211,4:SYS58732:PRINTH$(PC);
4210 PRINT"■■■■";H$(A);"■■■■";H$(X);"■■■";
4220 PRINT MID$(STR$(C),2,1);"■■■";MID$(STR$(N),2,1);"■■■";
4230 PRINT MID$(STR$(Z),2,1);"■■■■■";
4240 IF T THEN PRINT"BE ":RETURN
4250 PRINT"KI ":RETURN
4297 REM ******************
4298 REM UTASITAS KIVITELE
4299 REM ******************
4300 FOR P=0 TO 09:P1$(P)=P1$(P+1):P2$(P)=P2$(P+1):NEXT P
4310 P1$(10)=H1$+"    "+H2$+H3$+"   "
4320 P2$(10)=H4$+"   "+H3$
4330 POKE214,12:POKE211,0:SYS58732
4340 FOR P=0 TO 10:PRINT"■";P1$(P);"■■";P2$(P):NEXTP:RETURN
4397 REM ********************
4398 REM ARGUMENTUM ATVETELE
4399 REM ********************
```

```
4400 I1=ASC(MID$(IN$,P0,1)):I2=ASC(MID$(IN$,P0+1,1)):EF=0
4410 IF NOT (I1>47ANDI1<58ORI1>64ANDI1<71)THEN EF=1:RETURN
4420 IF NOT (I2>47ANDI2<58ORI2>64ANDI2<71)THEN EF=1:RETURN
4430 I1=I1-48:IF I1>9 THEN I1=I1-7
4440 I2=I2-48:IF I2>9 THEN I2=I2-7
4450 O=I1*16+I2:RETURN
4497 REM ****
4498 REM HIBA
4499 REM ****
4500 POKE214,8:POKE211,36:SYS58732
4510 PRINT"???";:POKE198,0:WAIT198,1:GETIN$
4520 PRINT"■■■   ";:GOTO 2000
4597 REM **********************
4598 REM ALSO KEPMEZO TORLESE
4599 REM **********************
4600 POKE214,12:POKE211,0:SYS58732
4610 FOR I=0 TO 10:P1$(I)="                    ":P2$(I)="       ":P3$(I)="
4620 PRINT"■";P1$(I);"■■";P2$(I);"■■■■";P3$(I):NEXTI:GOTO 2000
7997 REM **********************
7998 REM HEXADECIMALIS TABLAZAT
7999 REM **********************
8000 DATA "00","01","02","03","04","05","06","07","08","09"
8005 DATA "0A","0B","0C","0D","0E","0F"
8010 DATA "10","11","12","13","14","15","16","17","18","19"
8015 DATA "1A","1B","1C","1D","1E","1F"
8020 DATA "20","21","22","23","24","25","26","27","28","29"
8025 DATA "2A","2B","2C","2D","2E","2F"
8030 DATA "30","31","32","33","34","35","36","37","38","39"
```

```
8035 DATA "3A","3B","3C","3D","3E","3F"
8040 DATA "40","41","42","43","44","45","46","47","48","49"
8045 DATA "4A","4B","4C","4D","4E","4F"
8050 DATA "50","51","52","53","54","55","56","57","58","59"
8055 DATA "5A","5B","5C","5D","5E","5F"
8060 DATA "60","61","62","63","64","65","66","67","68","69"
8065 DATA "6A","6B","6C","6D","6E","6F"
8070 DATA "70","71","72","73","74","75","76","77","78","79"
8075 DATA "7A","7B","7C","7D","7E","7F"
8080 DATA "80","81","82","83","84","85","86","87","88","89"
8085 DATA "8A","8B","8C","8D","8E","8F"
8090 DATA "90","91","92","93","94","95","96","97","98","99"
8095 DATA "9A","9B","9C","9D","9E","9F"
8100 DATA "A0","A1","A2","A3","A4","A5","A6","A7","A8","A9"
8105 DATA "AA","AB","AC","AD","AE","AF"
8110 DATA "B0","B1","B2","B3","B4","B5","B6","B7","B8","B9"
8115 DATA "BA","BB","BC","BD","BE","BF"
8120 DATA "C0","C1","C2","C3","C4","C5","C6","C7","C8","C9"
8125 DATA "CA","CB","CC","CD","CE","CF"
8130 DATA "D0","D1","D2","D3","D4","D5","D6","D7","D8","D9"
8135 DATA "DA","DB","DC","DD","DE","DF"
8140 DATA "E0","E1","E2","E3","E4","E5","E6","E7","E8","E9"
8145 DATA "EA","EB","EC","ED","EE","EF"
8150 DATA "F0","F1","F2","F3","F4","F5","F6","F7","F8","F9"
8155 DATA "FA","FB","FC","FD","FE","FF"
8197 REM ******************
8198 REM UTASITAS-TABLAZAT
```

```
8193 REM ******************
8200 DATA "ASL  ",10,"CLC  ",24,"DEX  ",202,"INX  ",232,"LSR  ",74,"RTS  ",96
8201 DATA "SEC  ",56,"TAX  ",170,"TXA  ",138
8203 DATA "ADC #",105,"ADC $",101,"AND #", 41,"AND $", 37
8205 DATA "BCC $",144,"BCS $",176,"BEQ $",240
8215 DATA "BMI $", 48,"BNE $",208,"BPL $", 16
8220 DATA "DEC $",198,"EOR #", 73,"EOR $", 69
8225 DATA "INC $",230,"JMP $", 76,"JSR $", 32
8230 DATA "LDA #",169,"LDA $",165,"LDX #",162,"LDX $",166
8235 DATA "ORA #",  9,"ORA $",  5
8240 DATA "SBC #",233,"SBC $",229
8245 DATA "STA $",133,"STX $",134
READY.
```

KÜLÖNLEGES JELEK

A szövegben előforduló "↑" a Commodore billentyűzetén a RESTORE-tól balra, a RETURN felett lévő billentyűt jelenti. Némelyik nyomtató ehelyett csak egy " ^ "-jelet ír.

Némelyik programlistában a C=-jel is előfordul. Ezt ugyanúgy be kell gépelnünk, mert különben a képernyőn nem azt az eredményt kapjuk, amit vártunk.

Tár foglaltsági (memória) térkép

0	a processzorport adatirány-regisztere
1	a processzorport adatregisztere
2	nem használt
3/4	lebegőpontos-fixpontos átalakítás vektora
5/6	fixpontos-lebegőpontos átalakítás vektora
7	keresőjel
8	idézőjel-üzemmód jelző (flag)
9	TAB-oszlop
10	0, ha az utolsó utasítás LOAD: 1, ha VERIFY
11	beviteli puffer/diemenzió mutató
12	DIM-jelző (flag)
13	változótípus: FF=füzér, 00=számjegy
14	változótípus: 80=egész, 00=valós
15	idézőjel-üzemmód LIST-utasítás
16	FNx-jelző (flag)
17	Bevitel: 00=INPUT, 40=GET, 98=READ
18	előjel ARCTAN-nál utolsó összehasonlítás: 1=nagyobb, 2=egyenlő, 4=kisebb
19	aktuális file v. I/O-egység
20/21	egész szám, pl. címek, FRE (0)
22	füzér-veremtároló mutatója
23/24	az utolsó füzér mutatója
25–33	füzér-veremtároló
34–37	különféle mutatók
38–42	aritmetikai regiszter
43/44	BASIC-kezdőcím mutatója
45/46	változótartomány-kezdőcím mutatója
47/48	tömbök kezdőcímének mutatója
49/50	tömbök végének mutatója
51/52	füzérek kezdőcímének mutatója
53/54	füzér segédmutató
55/56	BASIC-RAM végének mutatója
57/58	az aktuális BASIC-sor száma

59/60	előző BASIC-sor száma
61/62	a következő utasítás mutatója CONT-nál
63/64	az aktuális DATA-sor száma
65/66	a következő DATA-elem mutatója
67/68	az utolsó DATA/INPUT/GET mutatója
69/70	az aktuális változó neve (2 betű)
71/72	az aktuális változó mutatója
73/74	az aktuális FOR ... NEXT-változó mutatója
75/76	a BASIC-programmutató segédregisztere
77	összehasonlító műveletek segédregisztere
78/79	az FNx mutatója
80/83	füzér-műveletek segédregisztere
84/86	függvények ugrási vektora
87/91	3-as aritmetikai akku
92/96	4-es aritmetikai akku
97/101	1-es aritmetikai akku
102	1-es akku előjele
103	a polinomkiértékelés számlálója
104	1-es akku kerekítő byte-ja
105–109	2-es aritmetikai akku
110	2-es akku előjele
111	1-es és 2-es akku összehasonlító regisztere
112	kerekítőbyte
113–114	a polinomiértékelés mutatója
115–138	CHRGET-rutin, a következő byte kiolvasása a BASIC-programból
122/123	BASIC-programmutató
139/143	utolsó RND-érték
144	állapotbyte (ST változóként)
145	1-es billentyűzetoszlop jelzője (flag)
146	időállandó kazetta üzemmódhoz
147	0=LOAD, 1=VERIFY
148	IEC-busz jelzője (flag)
149	jel az IEC-busznak (IEC-busz kiviteli puffer)
150	szalagvég-jelző (flag)
151	regiszterek közbülső tárolója
152	nyitott file-ok száma
153	aktuális beviteli egység (billentyűzet=0)
154	aktuális kiviteli egység (CMD, normál e.: =3)
155	paritásbyte kazetta-üzemmódban

156	byte-vétel jelzője (flag)
157	kivitel üzemmódja (128=közvetlen, 0=program)
158	kazetta üzemmód ellenőrző összeg
159	kazetta üzemmód hibajavítás
160–162	óra
163	soros kivitel bitszámlálója
164	szalagszámláló
165	számláló szalagra írásnál
166	mutató a kazettapufferben
167–171	szalagüzemmód jelzői
172/173	a kazettapuffer mutatója
174/175	program vége mutató (LOAD/SAVE)
176/177	időállandó a kazettának
178/179	a kazettapuffer kezdőcímének mutatója
180	bitszámláló (kazetta)
181	az RS 232 következő bitje (adás)
182	kiviteli byte
183	a file-név hossza byte-okban
184	az aktuális logikai file-szám
185	az aktuális másodlagos cím
186	az aktuális készülékszám (pl.: 8=floppy)
187/188	a file-név mutatója
189	soros kivitel segédregisztere
190	blokkszámláló (kazetta)
191	soros kiviteli puffer
192	kazettamotor állapotjelző (flag)
193/194	LOAD/SAVE kezdőcím
195/196	LOAD/SAVE végcím
197	lenyomott billentyű
198	a lenyomott billentyűk száma a pufferban
199	RVS-jelző (flag)
200	sor vége mutató (bevitel)
201/202	kurzor mutató bevitelkor (sor, oszlop)
203	lenyomott billentyű
204	kurzor állapotjelző (0=be)
205	kurzorvillogás számláló
206	a kurzor alatti jel
207	villogásjelző (flag)
208	jelző: bevitel a billentyűzetről vagy a képernyőről

209/210	az aktuális képernyősor mutatója
211	a kurzor oszlopa
212	kurzor programból / közvetlenül
213	képernyősor hossza (40/80)
214	a kurzor sora
215	utolsó billentyű
216	inzertek száma
217–242	a képernyősor kezdetének felső byte-ja
243/244	kurzorpozíció a szín-RAM-ban
245/246	a billentyűzet dekódertáblázat mutatója
247/248	az RS 232 beviteli puffer mutatója
249/250	az RS 232 kiviteli puffer mutatója
251/254	szabad byte-ok az operációs rendszer számára
255	BASIC-tár kezdete (* 64)
256/511	processzor-verem
256/266	formátum-átalakítás közbülső tára
256/318	szalaghibák javítása
512–600	BASIC beviteli puffer
601–610	logikai file-számok
611–620	készülékszámok
621–630	szekundercímek
631–640	billentyűzet puffer
641–642	BASIC-RAM kezdőcímének mutatója
643/644	BASIC-RAM végcímének mutatója
645	időzítési hibák a soros buszon, jelző (flag)
646	aktuális jelszín
647	kurzor alatti szín
648	video-RAM báziscímének felső byte-ja
649	billentyűzet puffer max. hossza
650	ismétlés-jelző (flag); 0=normál, 128=minden, 127=kikapcsolva
651	ismétlési sebesség számlálója
652	ismétlési késleltetés számlálója
653	SHIFT, C=, CTRL-jelző (flag)
654	ua., mint 653
655/656	a billentyűzet dekódertáblázat mutatója
657	SHIF/C=-billentyűkombináció kiiktatása, jelző
658	gördítés-jelző (flag)
659	az RS 232 ellenőrző regisztere

660	az RS 232 utasításregisztere
661/662	bit-idő
663	az RS 232 állapotregisztere
664	az átadott bitek száma (RS 232)
665/666	az RS 232 átviteli sebessége
667	az RS 232 által fogadott byte mutatója
668	az RS 232 beviteli mutatója
669	az RS 232 által továbbítandó byte mutatója
670	az RS 232 kiviteli mutatója
671/672	IRQ közbülső tár szalag üzemmódban
673	CIA 2 NMI-kapcsoló (flag)
674	CIA 1 A-időzítő
675	CIA 1 megszakítás-kapcsoló (flag)
676	A-időzítő kapcsolója (flag)
677	képernyősor
678–767	szabad RAM-terület
704–766	11-es sprite-blokk
768/769	hibaüzenet mutatója
770/771	BASIC melegindítás mutatója
772/773	szöveg-kód átalakítás mutatója
774/775	kód-szöveg átalakítás mutatója
776/777	az utasításvégrehajtás mutatója
778/779	a kifejezés kiértékelés mutatója
780	SYS akku
781	SYS X-regiszter
782	SYS Y-regiszter
783	SYS P-regiszter
784–787	USR-ugrás (cím a 785/786 rekeszekben)
788/789	hardver-megszakítás mutatója
790/791	BRK-megszakítás mutatója
792/793	NMI mutatója
794/795	OPEN mutatója
796/797	CLOSE mutatója
798/799	jelbevitel mutatója
800/801	jelkivitel mutatója
802/803	csatorna törlésének mutatója
804/805	INPUT mutatója
806/807	OUTPUT mutatója
808/809	STOP-billentyű lekérdezés mutatója

810/811	GET mutatója
812/813	minden csatorna zárása (mutató)
814/815	felhasználói – IRQ mutatója
816/817	LOAD mutatója
818/819	SAVE mutatója
820–827	szabad RAM-terület
828–1019	kazetta puffer
832–894	13-as sprite-blokk
896–958	14-es sprite-blokk
960–1022	15-ös sprite-blokk
1023	szabad
1024–2023	video-RAM
2024–2039	szabad
2040–2047	sprite-mutatók
2048–40960	BASIC-tároló
8192–16192	bit-térkép a nagyfelbontású grafikához
40960–49151	BASIC-értelmező – ROM
49152–53247	4k RAM gépi kódú programok számára
53248–57343	jelkészlet – ROM (jelgenerátor)
53248–53294	VIC-regiszterek
53295–54271	977 üres byte
54272–54300	SID-regiszterek
54301–55295	995 üres byte
55296–56295	szín-RAM
56296–56319	24 üres byte
56320–56335	CIA 1 regiszterei
56320/56321	billentyűzet lekérdezés és botkormány
56336–56575	240 byte üres
56576–56591	CIA 2 regiszterei
56577/56579	User-port-regiszter
56592–57343	752 üres byte
57334–65535	operációs rendszer ROM-ja

MELLÉKLET

```
1 REM *** M 15 ***
2 REM
10 REM PARAMETEREK MEGADASA
15 INPUT"XA,XE,YA,YE,SZIN,UZEMMOD(0/1)";XA,XE,YA,
   YE,SZ,L
20 REM
100 REM GRAFIKA BEKAPCSOLASA
101 REM ---------------------
120 POKE 53265,59
130 REM POKE 53270,216 (TOBBSZINU UZEMMOD BE)
140 POKE 53272,24
150 FOR I=8192 TO 16191:POKEI,0:NEXT
160 FOR I=1024 TO 2023:POKE I,4*16+1:NEXT
200 GOSUB 61100:WAIT198,255
61000 REM PONT RAJZOLASA ES TORLESE
61001 REM -------------------------
61010 Y=199-Y:IF Y<0 OR Y>199 THEN RETURN
61020 IF X<0 OR X>319 THEN RETURN
61030 X1=INT(X/8):X2=INT(Y/8)
61035 AD=8192+8*X1+320*X2+(YAND7)
61040 X3=2↑(7-(XAND7)):CA=1024+X1+40*X2
61050 POKECA,(PEEK(CA)AND15)OR16*SZ
61060 IF L=1 THEN POKEAD,PEEK(AD)AND(255-X3):RETURN
61070 POKEAD,PEEK(AD)ORX3:RETURN
61100 REM EGYENES RAJZOLASA
61101 REM -----------------
61110 IF ABS(XE-XA)<ABS(YE-YA)THEN 61160
61120 SP=(YE-YA)/ABS(XE-XA+1E-20):YK=YA
61130 FOR XX=XA TO XE STEP SGN(XE-XA)
61140 YK=YK+SP:Y=INT(YK+.5):X=XX:GOSUB61000
61150 NEXTXX:RETURN
61160 SP=(XE-XA)/ABS(YE-YA+1E-20):XK=XA
61170 FOR XX=YA TO YE STEP SGN(YE-YA)
61180 XK=XK+SP:X=INT(XK+.5):Y=XX:GOSUB 61000
61190 NEXTXX:RETURN
READY.
```

```
1 REM *** M 16 ***
2 REM
10 INPUT"KOZEPP.(X/Y),SUGAR,SZIN,UZEMMOD";XA,YA,R,SZ,L
20 GOSUB100:GOSUB61200:WAIT198,255:GOSUB200:END
30 REM
40 REM
100 REM GRAFIKA BEKAPCSOLASA
101 REM --------------------
120 POKE 53265,59
130 REM POKE 53270,216 (TOBBSZINU UZEMMOD BE)
140 POKE 53272,24
150 FOR I=8192 TO 16191:POKEI,0:NEXT
160 FOR I=1024 TO 2023:POKE I,SZ*16+1:NEXT
170 RETURN
175 REM
200 REM GRAFIKA KIKAPCSOLASA
201 REM --------------------
210 POKE 53265,155:REM GRAFIKUS UZEMMOD KI
220 REM POKE 53270,8 (TOBBSZINU UZEMMOD KI)
230 POKE 53272,21:REM NAGYBETUS JELKESZLET BE
240 RETURN
245 REM
61000 REM PONT RAJZOLASA ES TORLESE
61001 REM --------------------------
61010 Y=199-Y:IF Y<0 OR Y>199 THEN RETURN
61020 IF X<0 OR X>319 THEN RETURN
61030 X1=INT(X/8):X2=INT(Y/8)
61035 AD=8192+8*X1+320*X2+(YAND7)
61040 X3=2↑(7-(XAND7)):CA=1024+X1+40*X2
61050 POKECA,(PEEK(CA)AND15)OR16*SZ
61060 IF L=1 THEN POKEAD,PEEK(AD)AND(255-X3):RETURN
61070 POKEAD,PEEK(AD)ORX3:RETURN
61150 NEXTXX:RETURN
61155 REM
61200 REM KOR RAJZOLASA
61201 REM --------------
61210 FOR XX=0 TO R*0.7
61220 YY=INT(SQR(1-(XX/R)↑2)*R)
61230 X=XA+XX:Y=YA+YY:GOSUB61000
```

```
61240 X=XA+XX:Y=YA-YY:GOSUB61000
61250 X=XA-XX:Y=YA-YY:GOSUB61000
61260 X=XA-XX:Y=YA+YY:GOSUB61000
61270 X=XA+YY:Y=YA+XX:GOSUB61000
61280 X=XA+YY:Y=YA-XX:GOSUB61000
61290 X=XA-YY:Y=YA-XX:GOSUB61000
61300 X=XA-YY:Y=YA+XX:GOSUB61000
61310 NEXTXX:RETURN
READY.
```

```
1 REM *** M 24 ***
2 REM
10 POKE211,10:POKE214,24:SYS58732:REM KRZ.POZIC.
20 PRINT"◆":REM JEL A MEGADOTT HELYEN
25 GETA$:IFA$=""THEN 25:REM A KRZ. NEM LATHATO
30 POKE204,0:REM KURZOR BEKAPCSOLASA
35 PRINT"A$=";
40 GETA$:IFA$=""THEN40:REM A KRZ.LATHATO
50 PRINT A$:GOTO35
READY.
```

```
1 REM *** M 25 ***
2 REM
10 FOR I=1 TO 20
20 POKE199,1:REM INVERZ BE
30 PRINT"□* FELIRAT *":FORX=1TO100:NEXTX
40 POKE199,0:REM INVERZ KI
50 PRINT"□* FELIRAT *":FORX=1TO100:NEXTX
60 NEXT I
70 END
READY.
```

```
1 REM ***      M 26      ***
2 REM
3 REM BILLENTYU LEKERDEZES
4 REM ---------------------
5 REM "@"-BILLENTYU (OSZLOPKOD=223;BIT=6)
10 GETA$:IFA$=""THEN10
20 POKE56334,PEEK(56334)AND254
30 POKE788,52
40 POKE56320,223:P=PEEK(56321)
50 IF(PAND2↑6)=0 THEN PRINT"BILLENTYU LENYOMVA":
   GOTO 70
60 PRINT"EZ NEM AZ!"
70 POKE56334,PEEK(56334)OR1
80 POKE788,49
READY.
```

```
1 REM *** M 27 ***
2 REM
10 DATA 1,2,3,4,5
20 DATA 6,7,8,9,0
30 PRINT"A(I)";
40 FOR I=1 TO 10
50 READ A(I):PRINT A(I);
60 NEXT I
70 REM RESTORE
71 REM *******
80 POKE63,10:POKE64,0:REM 1.SZAM,2.SZAM
90 POKE65,41:POKE66,3:REM 3.SZAM,4.SZAM
100 REM
110 PRINT:PRINT"B(I)";
120 FOR I=1 TO 5
130 READ B(I):PRINT B(I);
140 NEXT I
150 END
READY.
```

Jegyzetek

Jegyzetek

Ára: 120,– Ft

SZÁMÍTÁSTECHNIKA A KÖNYVESBOLTOKBAN

NOVOTRADE – 2 C ÁRUHÁZ
1136 Bp., Balzac u. 35. Tel.: 402-954

ÁLLAMI KÖNYVTERJESZTŐ V. – NOVOTRADE 2C

BUDAPEST

Táncsics Könyvesbolt
1073 Lenin krt. 17.
Telefon: 422–178

BUDAPEST

Műszaki Könyváruház
1061 Liszt Ferenc tér 9.
Telefon: 420-353

MŰVELT NÉP KÖNYVTERJESZTŐ V. – NOVOTRADE 2C

PÉCS

Zrínyi Miklós Könyvesbolt
7621 Jókai u. 25.
Telefon: 72-12835

DEBRECEN

Szak- és ismeretterjesztő Könyváruház
4024 Hunyadi u. 8.
Telefon: 52-23237

SZOMBATHELY

Savaria Könyvesbolt
9700 Mártírok tere 1.
Telefon: 94-12341

VESZPRÉM

Kölcsey Ferenc Könyvesbolt
8200 Cserhát út 7.

BÉKÉSCSABA

Radnóti M. könyvesbolt
5600 Tanácsköztársaság út 2.
Telefon: 25-207

GYŐR

Pattantyús Á. Géza Szakkönyvesbolt
9021 Molnár Ferenc u. 9.

KECSKEMÉT

Művelt Nép Könyváruháza
6000 Március 15. u.
Széchenyiváros
Telefon: 06-76-28157

SZEGED

Tömörkény Könyvesbolt
6720 Lenin krt. 48.
Telefon: 62-21453

SZOLNOK

Szigligeti Könyvesbolt
5000 Ságvári krt. 35.
Telefon: 56-11133

MISKOLC

Chip-kuckó
3530 Tanácsház tér 14.