VIDEOTON TV-COMPUTER

GRAFIKA ÉS HANG PROGRAMOZÁSA

GÉPI KÓDBAN

Valójában assembly-ben, amit majd az assembler fordító alakít át gépi kódra ;-)

2019. március 11. – **befejezetlen változat**

Bertók Zsolt, 2019

Tartalomjegyzék

1. Alapok

Bár ennek a dokumentációnak a célja a *Videoton TV-Computer* grafika gépi kódú programozásának ismertetése *(amiről az alcímből már tudhatjuk, hogy valójában nem gépi kód lesz, hanem assembly :))*, ezzel együtt a *Z80 processzor* felépítését, regisztereinek és utasításkészletének ismertetését nem tartalmazza, hiszen az számos más forrásból megismerhető, például a ***Gépi kódú programozás kezdőknek*** című eredeti *TV-Computeres* könyvből.

Eredeti *TV-Computeres* könyvek elérhetők a **tvc.hu** weboldalon **pdf** formátumban. Ugyanitt megtalálhatóak a letölthető TVC emulátorok is (A **WinTVC** és a **PCZ80TVC** emulátorok akár 64 bites *Windows 10* alatt is működnek).

De akkor vágjunk bele!

A **Videoton TV-Computer** (továbbiakban **TVC**) videomemóriájának mérete **16 KByte**, ami a háromféle grafikus módban, háromféle felbontást és a hozzájuk tartozó színmélységet tartalmaz.

Grafikus mód	Felbontás	Színmélység	pixel / byte	VRAM mérete
Graphics 2	512 x 240 pixel	2 szín	8	(512x240)/8
Graphics 4	256 x 240 pixel	4 szín	4	(256x240)/4
Graphics 16	128 x 240 pixel	16 szín	2	(128x240)/2

A videomemória (továbbiakban **VRAM**) hasznos mérete minden felbontásban **15 360 byte**, ahogy a fenti táblázat utolsó oszlopában levő képlet mutatja. Azaz a 16 KByte-ból marad egy kicsi, amit nem jelenít meg a TVC az alapértelmezetten **240** soros felbontás miatt (az ismert oka, hogy legyen alul-felül elegendő méretű keret minden típusú CRT TV-n).

Mennyi is ez a maradék?

16 384 – **15 360** = **1024** byte (1 KByte – 16 *64 byte) – *ezt például vertikális hardver scroll-nál lehet okosan kihasználni. Talán oda is eljutunk egy következő részben, hogy ezt megmutassam :)*

Egy képernyősor minden felbontásban **64 byte**. Ez egy nagyon fontos információ, aminek nagy hasznát vesszük majd. Ezt a felbontás első számából (*képernyő szélessége pixelben*) és a táblázat *pixel / byte* értékéből láthatjuk. Ez minden felbontásban ennek a kettőnek a hányadosa:

(512 / 8) = (256 / 4) = (128 / 2) = **64**

A **VRAM** látható méretét ebből is kiszámolhatjuk: ha tudjuk, hogy egy képernyősor **64 byte** és minden felbontás **240** sorból áll, hiszen akkor:

64 * 240 = 15 360 byte

*Oké, már tudjuk, hogy mekkora a **VRAM**, de hogy férünk hozzá? Lássuk a következő fejezetben!*

2. A videomemória (VRAM) elérése

Ahhoz hogy a **VRAM**-hoz hozzáférjünk, be kell azt lapozni, mivel a rendszer felállásakor a videomemória nem érhető el. Ekkor a gép 4 x 16 Byte-ja a következő felállású: **U0, U1, U2, SYS.** E helyett mi az **U0, U1, VID, SYS** állapotot szeretnénk bekapcsolni, ahol a **VID** a **VRAM**-ot jelöli.

U0-U3	**U**SER (felhasználói) memória
SYS	**SYS**TEM – a TVC ROM a BASIC-el
VID	**VID**EORAM (**VRAM**)

*Részleteket, további lehetőségeket, lapozási kódokat lásd az **Operációs rendszer** c. könyvben.*

A **VRAM** belapozást mindössze 3 sornyi assembly utasítással meg tudjuk tenni:

```
LD   A,$50     ; lapozási kód: U0, U1, VID, SYS ($50 = decimális 80)
LD   ($3),A    ; kód kiírása a P_SAVE rendszerváltozóba (00003-as cím)
OUT  ($2),A    ; lapozási kód kiküldés a 2-es portra
```

Megjegyzés: ***P_SAVE** csak akkor működik, ha az **U0** van belapozva (Vass Sándor tapasztalata). Gondolom ez azért van így, mivel a **$3**-as memóriacím (**P_SAVE**) az **U0**-án helyezkedik el, így ha az nincs belapozva, akkor ott egyszerűen nem a **P_SAVE** rendszerváltozó található.*

A belapozást követően a decimális **32 768 – 49 151** (hexa: **$8000-$BFFF**) memóriacímen lehet elérni a **VRAM**-ot. Ez azt jelenti, hogy ha bármit erre a területre írunk, akkor az valamit meg fog jeleníteni a képernyőn. Kivéve az utolsó 1 KByte-ot, ami ugye alapértelmezetten nem látszik a képernyőn.

De elég a rizsából, lássunk már végre valami grafikát! :) Lapozz!

A **VRAM**-ba írást akár BASIC-ben is kipróbálhatjuk és máris pixeleket varázsolunk a képernyőre.

Először lapozzuk be a **VRAM**-ot:

POKE 3,80: **OUT** 2,80

Aztán rakjunk ki egy 4 pixeles vonalat a képernyő közepére:

POKE 40480, 15

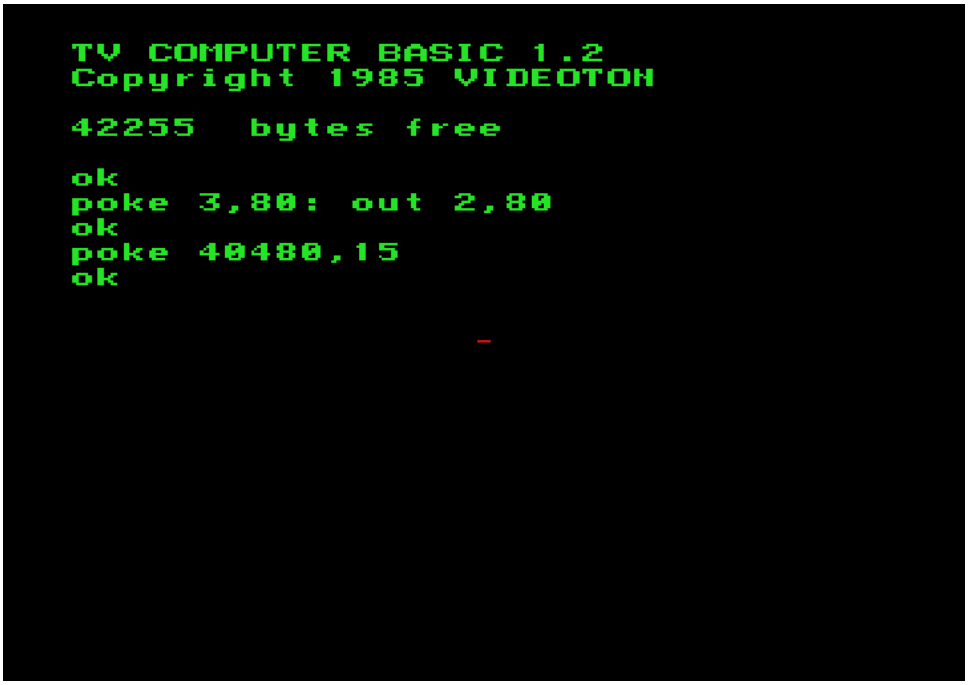
32768 + 120*64 + 32 = **40480** → 120. sor közepe

FIGYELEM! *Igazából ez csak emulátorban (WinTVC-n) működik, mert igazi TVC-n a BASIC parancssori értelmező lapozgatja a memóriát bőszen, így mi hiába lapozzuk be a* ***VRAM****-ot, az átlapozza másképp :) De azért nincs nagy baj, így is elérjük a* ***VRAM****-ot, csak* ***16 KByte-tal*** *arrébb, mivel az ilyenkor nem a* ***32768****-as címen kezdődik, hanem* ***49152****-esen.*

Tehát a második poke után írandó érték a következőképpen alakul:

49152 *+ 12*64 + 32 =* ***56864*** *→ ez lesz igazi vason a képernyő közepe*

Éééééééééés ott egy piros vonal a képernyő közepén! :) Nyugi, a következő fejezetben assembly-ben is megcsináljuk!



A **VRAM** a képernyő bal felső sarkában kezdődik, képernyősoronként **64 byte**-ot foglal el **240** soron keresztül , folyamatosan, fentről le. Valójában **256** sor (256 * 64 byte = 16 384 byte = 16 KByte), de az utolsó **16** sor, azaz 1 KByte (16 * 64 byte), alapból nincs megjelenítve, ahogy erről már beszéltünk. A **$8000** cím a képernyő bal felső byte-jának a címe, a **$BBFF** pedig a jobb alsó, utolsó byte-é. **$BFFF**, a **VRAM** vége, a nem látható 255. sor utolsó byte-ja.

*Melléklet az Operációs rendszer c. könyvből a **VRAM** felépítéséről*

8. Videomemória

tv sor	"Karakter" 0	1	2	3	- - -	61	62	63	"Kar." sor
0	8000	8001	8002	8003	- - -	803D	803E	803F	
1	8040	8041	8042	8043	- - -	807D	807E	807F	
2	8080	8081	8082	8083	- - -	80BD	80BE	80BF	0
3	80C0	80C1	80C2	80C3	- - -	80FD	80FE	80FF	
4	8100	8101	8102	8103	- - -	813D	813E	813F	
5	8140	8141	8142	8143	- - -	817D	817E	817F	
6	8180	8181	8182	8183	- - -	81BD	81BE	81BF	1
7	81C0	81C1	81C2	81C3	- - -	81FD	81FE	81FF	
8	8200	8201	8202	8203	- - -	823D	823E	823F	
9	8240	8241	8242	8243	- - -	827D	827E	827F	
10	8280	8281	8282	8283	- - -	82BD	82BE	82BF	2
11	82C0	82C1	82C2	82C3	- - -	82FD	82FE	82FF	
:	:	:	:	:		:	:	:	:
232	BA00	BA01	BA02	BA03	- - -	BA3D	BA3E	BA3F	
233	BA40	BA41	BA42	BA43	- - -	BA7D	BA7E	BA7F	
234	BA80	BA81	BA82	BA83	- - -	BABD	BABE	BABF	58
235	BAC0	BAC1	BAC2	BAC3	- - -	BAFD	BAFE	BAFF	
236	BB00	BB01	BB02	BB03	- - -	BB3D	BB3E	BB3F	
237	BB40	BB41	BB42	BB43	- - -	BB7D	BB7E	BB7F	
238	BB80	BB81	BB82	BB83	- - -	BBBD	BBBE	BBBF	59
239	BBC0	BBC1	BBC2	BBC3	- - -	BBFD	BBFE	BBFF	

Bár a ***Zilog Z80*** processzor, ami a **TVC**-ben van, egy *8 bites* mikroprocesszor, szerencsére vannak *16 bites* regiszterei is, amik segítségével meg tudjuk címezni a **VRAM** teljes területét. Általában a legoptimálisabb a **HL** vagy a **DE** regisztereket használni erre, de ez függ az adott feladattól és annak optimalizálhatóságától.

*Érdekesség: nem csak címzésre, de adatmozgatásra is kihasználhatjuk a Z80 processzor 16 bites regisztereit. Csodálatos trükk például, hogy a stack pointert (**SP** regisztert) a **VRAM** megfelelő címére állítva mondjuk egy **PUSH BC** utasítással egyszerre **2 byte**-ot tudunk kiírni a képernyőre, míg minden más esetben egy utasítással maximum csak 1 byte-ot. Persze előtte az **SP** regisztert el kell menteni, majd ha végeztünk, akkor visszaállítani és közben a **PUSH** / **POP** hagyományos módon nem használható! De közben kétszeres sebességgel tudjuk írni a VRAM-ot (meg persze a hagyományos memóriát is, ha az **SP**-t olyan memóriacímre állítottuk), hiszen egy 16 bites regiszterből tudunk adatot elhelyezni az adott címre, míg mondjuk egy „adatmozgató” utasítás, az **LD** vagy az **LDI** csak egy byte-ot, azaz 8 bitet mozgat. Persze ez csak akkor igaz, ha mindig ugyanazt az 2 byte-ot szeretnénk kiírni folyamatosan, például egy területet azonos színnel szeretnénk feltölteni vagy a képernyőt törölni. Egyéb esetekben a plusz adattöltő utasítások miatt, amikkel a 16 bites regisztert újra és újra feltöltjük a kirakandó adatokkal, már kétséges, hogy valóban gyorsabb lesz-e ez a módszer. Egy sprite kirakása esetén például már nem biztos, hogy fel tudjuk használni gyorsításra. De ha odaérünk a leírásban, kiderítjük! :) Mindenesetre köszi a tippet az Elite programozójának, akinek ez eszébe jutott és Major Tamásnak, aki ezt megtalálta az Elite kódjában a múlt században és tegnap megosztotta velem! :)*

De nézzük meg azt az esetet is, amikor éppen nincs szükségünk a **VRAM**-ra, viszont hozzá szeretnénk férni a 32K feletti – ***32 768**-as (hexa **$8000**) címtől kezdődő* – 16 Kbyte-nyi, szabadon használható memóriaterülethez, ahová ugye a **VRAM**-ot lapoztuk be. Akkor ehhez az eredeti memória lapozást vissza kell állítanunk. Ezt a következő módon tudjuk megtenni:

```
LD   A,$70      ; lapozási kód: U0, U1, U2, SYS
LD   ($3),A     ; kiírás a P_SAVE memória területre
OUT  ($2),A     ; kiküldés a 2-es portra
```

Hogy a gyakorlatban milyen módon tudunk assembly-ben programozni TVC-n, az kiderül a következő fejezetből.

3. Hogyan csináljuk a gyakorlatban?

Szuper, de hogy tudom én ezt kipróbálni, miben/mivel tudok assembly kódok írni, fordítani, futtatni? – kérdezheted jogosan. Én pedig megválaszolom :)

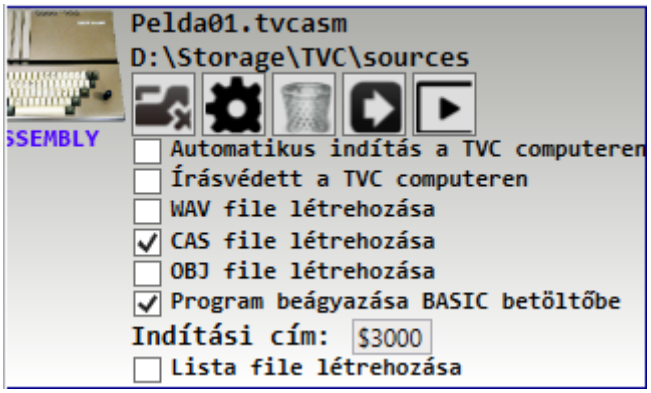
Windows alól erre a legbarátságosabb eszköz 2019-ben a TVC **Studio**, amit *István Oravecz* munkájának köszönhetően használhatunk. A keretrendszer tartalmaz egy jó szövegszerkesztőt, amiben én legjobban a *kódszekciókat* (*Visual Studio*t ismerőknek *region-ok*) szeretem, amik segítségével átláthatóbbá tehető egy hosszabb forráskód is. De nem csak szerkesztésre alkalmas, hanem *assembly* kódok lefordítására és BASIC betöltő részt tartalmazó **.CAS** állomány készítésére is, ami azonnal betölthető egy emulátorba és ott futtatható. Ezen felül **.wav** fájlt is képes készíteni a **.CAS** fájlból, ami közvetlenül a gépbe is betölthetővé teszi a programot. Szóval remek kis fejlesztőeszköz ez a *TVC Studio, köszönjük!*

*Itt jegyzem meg, hogy készíthető olyan assembly forrás, aminek a kezdőcímét (*org*) decimális **6639**-re – a BASIC terület kezdetére – állítva és a megfelelő BASIC tokeneket tartalmazó kódokat az elejére helyezve, majd lefordítva egy önmagában is működő.. .**cas** fájlt eredményez, amit nem kell BASIC betöltővel „bepoke-olni”, így ez hosszabb kódoknál működőképesebb és kényelmesebb megoldás. Erről is szót ejtek majd később.*

De akkor nézzük az előző fejezetben szereplő BASIC példát, a piros vonalat a képernyő közepén, a TVC Studio-ba beírható assembly szintaxissal:

```
ORG   $3000          ; a kódunk ide fog kerülni a memóriában betöltéskor
LD    A,$50          ; A regiszterbe a memória lapozási kód: U0, U1, VID, SYS
LD    ($3),A         ; lapozási kód kiírása P_SAVE rendszerváltozóba (00003-as címre)
OUT   ($2),A         ; és kiküldése a 2-es port-ra – a VRAM belapozása kész
LD    HL,32768+120*64+32 ; HL regiszterbe a cím: VRAM eleje + 120 sor + 32*4 pixel
LD    A,15           ; A =15 → 2. szín beállítása az összes pixelre a byte-ban
LD    (HL),A         ; A, azaz 15 kiírása a HL által mutatott VRAM címre
LD    A,$70          ; A regiszterbe az eredeti memória lapozási kód: U0, U1, U2, SYS
LD    ($3),A         ; lapozási kód kiírása P_SAVE rendszerváltozóba
OUT   ($2),A         ; és kiküldése a 2-es port-ra – eredeti lapozás visszaállítva
RET                  ; RETURN - visszatérés a hívóhoz, ez esetben a BASIC-be
END                  ; a program vége – jelzés a TVC Studionak
```

Az alábbi kép szerint beállított **TVC Studio**-val elkészített BASIC betöltős **.cas** fájl betölthető mondjuk a **WinTVC** vagy a **PCZ80TVC** emulátorokba és azokban futtatható.



4. Írás a képernyőre

*Azt ígértem, hogy ha **VRAM** területre írunk, akkor az meg fog jelenni a TVC képernyőjén. Lássuk!*

Ha a **VRAM** már belapozásra került (*lásd az előző fejezetben*), akkor az elejére írjunk ki egy byte-ot:

```
LD   HL,$8000  ; HL-be betöltjük a VRAM legelső byte-jának címét
LD   A,$FF     ; betöltünk az A regiszterbe 255-öt
LD   (HL),A    ; kiírjuk a VRAM első byte-jára az A regisztert, azaz 255-öt
```

A paletta 3. színe szerinti 4 pixel széles vonal fog megjelenni a bal felső sarokban. ***De miért?*** Alapértelmezetten **Graphics 4**-ben indul el a **TVC**. Ez a **256x240** pixel felbontású, **4 színű** grafikus mód. Ebben a módban **1 byte-on 4 pixel** kerül megjelenítésre, hiszen a 4 -féle szín 0-3 értékként kerül letárolásra, amihez 2 bit elegendő, 1 byte pedig 8 bit, ebből adódóan 8 / 2 = 4 (pixel).

1-1 pixel színét nem egymást követő bitek mutatják, hanem egy bit a byte felső 4 bitjéből és egy bit az alsó 4 bitjéből. Így néz ki 4 pixel 1 byte-on bitenként (*az azonos színek csak az azonos pixelhez tartozó biteket jelölik, nincs közük a pixelek tényleges színéhez!*):

Érték:	128	64	32	16	8	4	2	1
Bit sorsz.:	7. bit	6. bit	5. bit	4. bit	3. bit	2. bit	1. bit	0. bit
Pixel/szín:	**P1** Szin0	**P2** Szin0	**P3** Szin0	**P4** Szin0	**P1** Szin1	**P2** Szin1	**P3** Szin1	**P4** Szin1

P1-P4 az egy byte-on belüli pixeleket jelenti balról-jobbra, a **Szin0** az adott pixel színének alsó bitjét, a **Szin1** pedig a felső bitjét. Az alábbi táblázatokban a kék háttér **1,** a fehér **0** bit értéket jelöl.

Az 1. pixel 1. színre állításakor így kell a biteket beállítani (Szin0: 1 Szin1: 0) – a byte értéke = **128**

7. bit	6. bit	5. bit	4. bit	3. bit	2. bit	1. bit	0. bit
P1 Szin0	P2 Szin0	P3 Szin0	P4 Szin0	P1 Szin1	P2 Szin1	P3 Szin1	P4 Szin1

Az 1. pixel 2. színre állításakor így néz ki a byte (Szin0: 0 Szin1: 1) – a byte értéke = **8**

7. bit	6. bit	5. bit	4. bit	3. bit	2. bit	1. bit	0. bit
P1 Szin0	P2 Szin0	P3 Szin0	P4 Szin0	P1 Szin1	P2 Szin1	P3 Szin1	P4 Szin1

Az 1. pixel 3. színre állításakor így néz ki a byte (Szin0: 1 Szin1: 1) – a byte értéke: **136**

7. bit	6. bit	5. bit	4. bit	3. bit	2. bit	1. bit	0. bit
P1 Szin0	P2 Szin0	P3 Szin0	P4 Szin0	P1 Szin1	P2 Szin1	P3 Szin1	P4 Szin1

Az 1. pixel 1. színe és a 3. pixel 3. színre állításakor így néz ki a byte – a byte értéke: **162**

7. bit	6. bit	5. bit	4. bit	3. bit	2. bit	1. bit	0. bit
P1 Szin0	P2 Szin0	P3 Szin0	P4 Szin0	P1 Szin1	P2 Szin1	P3 Szin1	P4 Szin1

Mind a 4 pixel a 3. színre állításakor így néz ki a byte – a byte értéke: **255 (hexa: $FF)**

7. bit	6. bit	5. bit	4. bit	3. bit	2. bit	1. bit	0. bit
P1 Szin0	P2 Szin0	P3 Szin0	P4 Szin0	P1 Szin1	P2 Szin1	P3 Szin1	P4 Szin1

Nyilvánvalóan egy byte az összes -féle kombinációban tartalmazhatja a bitek ki-be kapcsolt állapotát, attól függően, hogy melyik pixelek milyen színnel vannak éppen megjelenítve. Ezekből csak néhányat mutattak be a fenti példák.

Az utolsó példa alapján, ha a képernyő-memória valamelyik byte-jába **255** értéket (hexa **$FF**) töltünk, akkor ez azt jelenti, hogy az1-4 pixelek összes bitjét 1-re állítjuk. Tehát az előző példaprogramunkban, ahol a **$8000**-es címre **$FF** értéket írtunk, nem tettünk mást, mint a képernyő első sorának első byte-jában mind a négy pixelt beállítottuk a paletta 3. színére, így rajzolva egy 4 pixel széles és 1 pixel magas vonalat a paletta 3. színével, ami a TVC bekapcsolása után a sötétkék szín.

Érték:	**128**	**64**	**32**	**16**	**8**	**4**	**2**	**1**
Bit sorsz.:	7. bit	6. bit	5. bit	4. bit	3. bit	2. bit	1. bit	0. bit
Pixel/szín:	**P1** Szin0	**P2** Szin0	**P3** Szin0	**P4** Szin0	**P1** Szin1	**P2** Szin1	**P3** Szin1	**P4** Szin1

Látható a táblázatból, hogy ha csak az **1. pixelt** szeretnénk **1. színre** állítani (**P1 Szin0 bit**), akkor **255** helyett **128**-at kell a címre írni. Ha a **2. pixelt** szeretnénk **1. színre** állítani (**P2 Szin0 bit**), akkor pedig **64**-et. A **3. pixel** esetén ez az érték **32**, míg a **4. pixel** esetén **16**. Ezek nem mások, mint megfelelő bit által mutatott 2-es számrendszerbeli értékek. Nem is nehéz, nem igaz? :) A **2. színhez** a pixeleknek megfelelő alsó biteket kell ugyanígy beállítani (**P1 Szin1, P2 Szin1, P3 Szin1, P4 Szin1 bitek**), a **3. színhez** pedig a pixeleknek megfelelő felső és az alsó biteket egyszerre.

Oké, de ha például **128**-at teszünk abba a byte-ba, akkor nem csak az **1. pixelt** állítjuk ezzel **1. színűre**, hanem a **2-3-4. pixeleket** is egyben **0. színűre**, hiszen a byte összes többi bitjében **0** van, ezért aztán felülírjuk a 2-3-4. pixeleket is. De mi van akkor, ha azokon a pixeleket már ki van valami rajzolva a képernyőre, amit nem szeretnénk felülírni, vagy nem 4 pixelt szeretnénk egyszerre kiírni, csak egyet? Erről szól a következő fejezet.

5. A VRAM pixelenkénti írása

*Az előző fejezetben kiderült, hogy a videomemóriát, hogy tudjuk egyszerűen byte-onként írni, de **Graphics 4** módban 1 byte-ban 4 pixel van, így ha byte-onként írjük a **VRAM**-ot, akkor mindig **4 pixel**t írunk felül, ami ritkán megfelelő megoldás.*

Hogy tudunk egyszerre csak 1 pixelt írni a VRAM-ba?

Úgy, hogy a byte-nak csak bizonyos bitjeit írjuk felül, nem az egészet. Tudom, te is erre gondoltál ;-)

Erre nekem kapásból két módszer jut eszembe:

I. Módszer – SET / RES

Van két bit manipuláló utasítás *Z80 assembly*-ben, amivel egy-egy bitet tudunk beállítani 1-re (**SET**), vagy 0-ra (**RES**). *A zöld megjegyzések elején szögletes zárójelekben a végrehajtási ciklus szerepel.*

```
LD   HL,$8000     ; [10] megcímezzük a VRAM első byte-ját
SET  7,(HL)       ; [15] a $8000-es címen levő byte 7. bitjét 1-re állítjuk
```

Egy bit beállításával azonban csak a szín felét tudjuk beállítani, de nekünk a másik bitet is be kell, hogy az adott pixel mindkét szín-bitjét módosítsuk. Így tehetjük ezt meg, ha az **1. pixelt 1. színre** akarjuk állítani:

```
LD   HL,$8000     ; [10] megcímezzük a VRAM első byte-ját
SET  7,(HL)       ; [15] a $8000-es címen levő byte 7. bitjét 1-re állítjuk
RES  3,(HL)       ; [15] a $8000-es címen levő byte 3. bitjét 0-ra állítjuk
```

Így pedig, ha a **3. pixelt** akarjuk a **2. színre** állítani:

```
LD   HL,$8000     ; [10] megcímezzük a VRAM első byte-ját
RES   5,(HL)      ; [15] a $8000-es címen levő byte 5. bitjét 0-ra állítjuk
SET  1,(HL)       ; [15] a $8000-es címen levő byte 1. bitjét 1-re állítjuk
```

II. Módszer – AND / OR

Logikai és /vagy műveletet hajtanak végre bitenként az **AND** / **OR** utasítások. Az **OR**-nak az lenne az előnye a **SET**-el szemben, hogy akár két bitet is be tudunk állítani egy utasítással (persze akár mind a 8-at is). Az **AND**-el viszont törölni tudunk, akár két bitet is egyszerre (természetesen ezzel is akár mind a 8 bitet tudjuk egy utasítással törölni). Szóval az **AND** / **OR** akkor lehet hasznos, ha egy byte-on belül több pixelt is szeretnénk egyszerre állítani, vagy törölni, illetve ha a **0.** vagy a **3.** színt akarjuk beállítani, mivel ekkor egyetlen **AND** (0. színhez) vagy egyetlen **OR** (3. színhez) utasítás is elég az egy pixelhez tartozó 2 darab szín-bit beállításához. *Egy **AND** / **OR** párossal, azaz 2 utasítással, akár 3 pixelt is be tudunk egyszerre állítani tetszőleges színre egy byte-on belül (nyilván 4 pixelt is, de akkor már egyszerűbb 1 utasítással kiírni 1 egész byte-ot :)). A **RES** / **SET** párosból ehhez 2 helyett 6 utasítása van szükség, de persze ez speciális eset.*

Lássuk! Az **1. pixelt** beállítjuk az **1. színre** (*tételezzük fel, hogy **HL** regisztert beállítottuk egy előző utasítássorban a megfelelő **VRAM** címre*) :

```
LD   A,(HL)    ; [ 7] A regiszterbe beolvasunk 1 byte-ot a VRAM-ból
OR   128       ; [ 7] A 7. bitjét beállítjuk → 128
AND  255-8     ; [ 7] A 3. bitet töröljük → mindenhol 1, kivéve a 3. bit (8)
LD   (HL), A   ; [ 7] visszaírjuk a módosított byte-ot a VRAM-ba
```

*Érdekesség: a végrehajtása összesen 4*7 = **28** ciklus, míg a **SET** / **RES** páros 2*15 = **30** ciklus.*

*Ha a 4. pixelt állítjuk a 3. színre, akkor az **AND**-et megspóroljuk, így már csak **21** ciklus*:

```
LD   A,17      ; [ 7] A 4. és 0. bitjét beállítjuk → 16 + 1 = 17
OR   (HL)      ; [ 7] A OR VRAM byte, amire HL regiszter mutat
LD   (HL), A   ; [ 7] visszaírjuk a módosított byte-ot a VRAM-ba
```

Ez persze csak egy pixel kirakása egy helyre, ennél izgalmasabb feladat, ha vonalat kell húzni, ahol az eltérő folyamat miatt másféle optimalizálást lehet elvégezni, szóba kerülhetnek újabb izgalmas utasítások, amikkel egyszerűsíthetjük az életünket. Erre nézünk példákat a következő fejezetben.

6. Vonal rajzolása

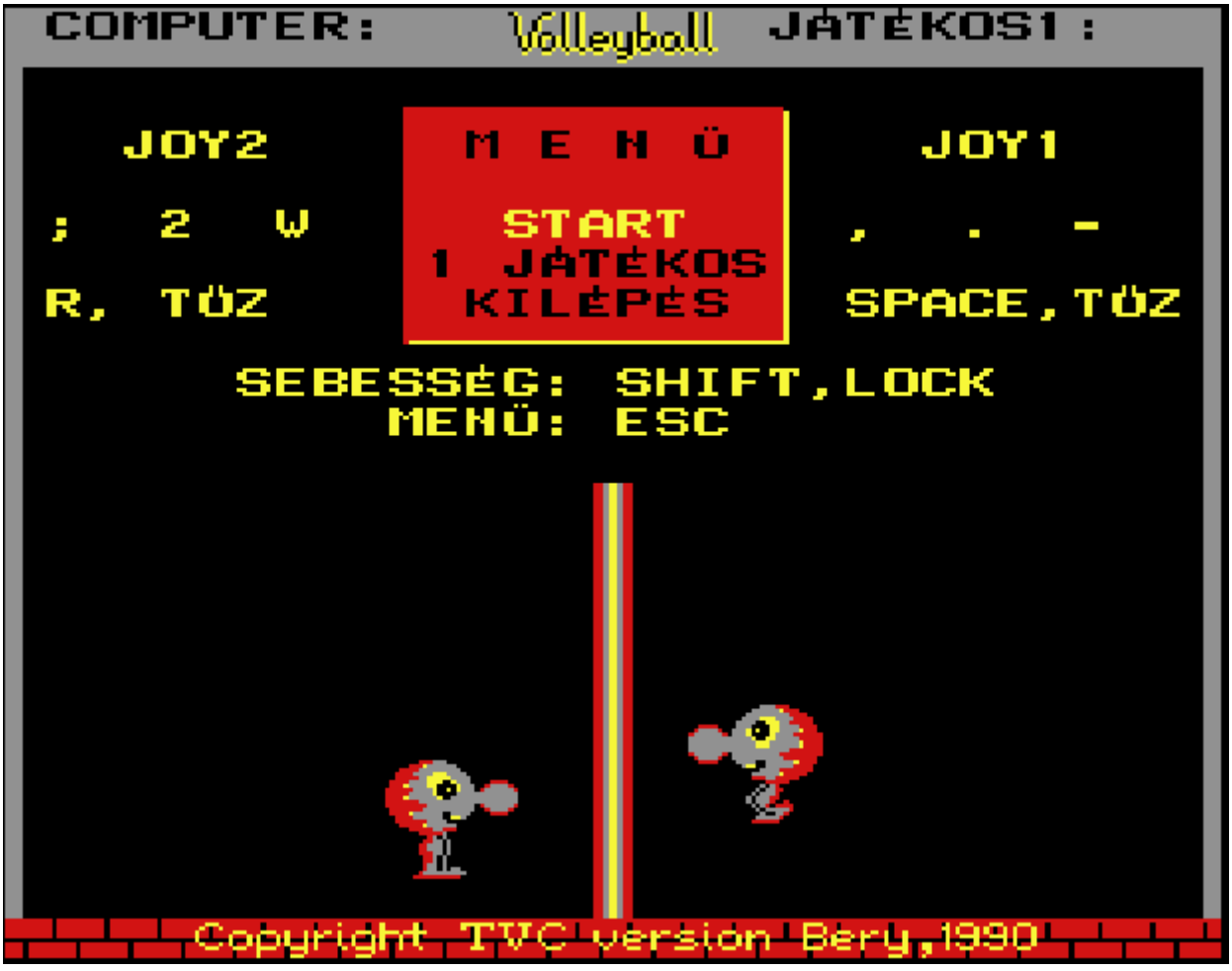
Majd egyszer ez is jön!

6. Sprite kirakása

Na, végre valami izgalmas dolog! :) Azonban kicsit füllentős ez a cím, mert elsőre igazából nem sprite-ot fogunk kirakni, hanem mondjuk úgy, hogy egy képdarabot. Azaz inkább egy négyszögletes figurát, ami annyiban tér el a hagyományos sprite-tól, hogy ugyan úgy néz ki, mint egy sprite, de nem átlátszó. Ez mit jelent? Azokon a pontjain, ahol nincs kirajzolandó pixele (a mintában fekete), ott kirakás után nem a hátteret fogjuk látni, hanem szépen felülírja azt a saját háttér színű pixelével.

Akkor miért fárasztalak ezzel a kirakással? Mert sokkal gyorsabb, mint ha egy háttérszíneiben átlátszó sprite-ot szeretnénk kirakni, és bizonyos esetekben nincs szükséged „igazi” sprite kirakásra, mert ahol a sprite-od mozog, ott nincs „rajzolt” háttér, csak egyféle háttérszín, így a nem átlátszó sprite-od nem fogja azt „elrontani”. *De nyugi, készítünk „rendes” sprite-kirakót is majd! :)*

Itt egy példa az általam 1990-ben elkövetett *Volleyball* c. játékból. A figurák fekete háttér előtt mozognak, így nincs szükség arra, hogy a háttérszínű (fekete) pixeleik átlátszóak legyenek, hiszen pont úgy néz ki a háttér is, így ha rárakjuk arra, azzal semmit nem takarunk le. Másik példának a *Kísértetkastély* c. játékban szereplő ellenfeleket is írhatom, amik szintén ezzel a technikával kerülnek ki a képernyőre. De minden olyan helyen érdemes ezt alkalmazni, ahol egyszínű a háttér.



De akkor lássuk a medvét*! Hogy néz ki egy ilyen négyzetes sprite-kirakó assembly-ben:

```
            ORG  $4000              ;erre a címre kerül a lefordított kód

            CALL VRAMON             ;meghívjuk VRAM belapozást
            ; beállítjuk a PutSprite szubrutin paramétereit:
            LD   HL,SPRITE          ;HL: a sprite kezdőcíme
            LD   DE,32768+106*64+30;DE: képernyő pozíció: 106. sor, 30. byte
            LD   A,28               ;A : a sprite magassága pixelsorokban
            CALL PUTSPRITE          ;meghívjuk a sprite-kirakót
            CALL VRAMOFF            ;meghívjuk a VRAM kilapozást
            RET                     ;visszatérünk a BASIC-be

            ;sprite-kirakó szubrutin. Paraméterek HL, DE és A regiszterekben
PUTSPRITE   LD   BC,64              ;BC = 64
            LDI                     ;HL címről 1 byte DE címre és DEC(BC)
            LDI
            LDI
            LDI                     ;ahány byte széles a sprite, annyi LDI
kell
            EX   DE,HL              ;DE <=> HL csere
            ADD  HL,BC              ;HL = HL + (64 - szélesség) → új sor
            EX   DE,HL              ;DE <=> HL csere vissza
            DEC  A                  ;csökkentjük a kiírandó sorok számát
            JP   NZ,PUTSPRITE;ha A nem 0, akkor folytatjuk a kiírást
            RET                     ;kész, visszatérés a hívóhoz

            ;VRAM belapozó szubrutin
VRAMON      LD   A,$50              ;lapozási kód: U0, U1, VID, SYS
            LD   ($3),A             ;kód kiírása a P_SAVE rendszerváltozóba
            OUT  ($2),A             ;lapozási kód kiküldés a 2-es portra
            RET                     ;visszatérés a hívóhoz

            ;VRAM kilapozó szubrutin
VRAMOFF     LD   A,$70              ;lapozási kód: U0, U1, VID, SYS
            LD   ($3),A             ;kód kiírása a P_SAVE rendszerváltozóba
            OUT  ($2),A             ;lapozási kód kiküldés a 2-es portra
            RET                     ;visszatérés a hívóhoz

            ;a sprite pixelei byte-okban (16x28 pixel → 4x28 byte)
SPRITE      DB    48,0,0,192
            DB    112,176,208,224
            DB    112,240,240,224
            DB    112,240,240,224
            DB    48,48,192,192
            DB    96,240,240,96
            DB    112,240,240,224
            DB    112,48,192,224
            DB    48,48,192,192
            DB    48,240,240,192
```

```
DB      112,192,48,224
DB      112,224,112,224
DB      48,176,208,192
DB      16,192,48,128
DB      2,112,224,4
DB      7,56,193,12
DB      7,12,3,14
DB      15,15,15,15
DB      9,14,7,9
DB      120,15,15,225
DB      112,14,7,224
DB      1,15,15,8
DB      32,0,0,64
DB      48,240,240,192
DB      16,240,240,128
DB      0,224,112,0
DB      0,14,7,0
DB      3,14,7,12
END                         ;a program vége – jelzés a TVC Studionak
```

A fenti forráskód egy az egyben beilleszthető a *TVC Studio*-ba. A BASIC betöltő és a **CAS** fájl elkészítése után *(Indítási cím: **$4000**)* már be is lehet tölteni emulátorba vagy igazi *TVC*-be.
Az eredménynek így kell kinéznie, Ő egy **16** pixel széles, **28** pixel magas *medve-*féleség*:

Mit kell tudnunk erről a sprite-kirakóról?

Ez a sprite-kirakó mindhárom grafikus módban módosítás nélkül használható a megfelelő sprite adatok (`SPRITE DB` sorok) megadása mellett. Ez ki is próbálható: betöltés után váltsunk át **Graphics 2**-re, majd futtassuk a programot, majd **Graphics 16**-ra és ismét futtassuk. A sprite kicsit „szélütötten” :), de mindkét módban meg fog jelenni. A furcsaság oka, hogy a sprite pixeladatai a **Graphics 4** módhoz lettek elkészítve, ott felelnek meg az eredeti kinézet pixeleinek, de ugyanezek a byte-ok más pixeleket eredményeznek más grafikus módokban. A következő két táblázat mutatja, hogy mi az eltérés a grafikus módok között az egy byte-on levő pixeleket tekintve.

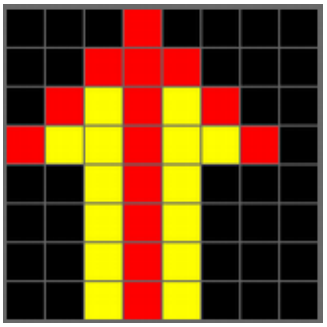
Graphics 2	**1 byte = 8 pixel**	a byte minden bitje **1** pixelnek felel meg
Graphics 4	**1 byte = 4 pixel**	a korábban ismertetett **2** bit / pixel felépítésű
Graphics 16	**1 byte = 2 pixel**	**4** bit / pixel felépítésű (I-G-R-B bitek)

```
   GRAPHICS  2  (8 pont/byte, pontsorrend:  1 2 3 4 5 6 7 8 )
┌──────┬──────┬──────┬──────┬──────┬──────┬──────┬──────┐
│  B7  │  B6  │  B5  │  B4  │  B3  │  B2  │  B1  │  B0  │
├──────┼──────┼──────┼──────┼──────┼──────┼──────┼──────┤
│1.p. C0│2.p. C0│3.p. C0│4.p. C0│5.p. C0│6.p. C0│7.p. C0│8.p. C0│
│  ..... .....  C1=0  ..... ..... .....  │
└──────┴──────┴──────┴──────┴──────┴──────┴──────┴──────┘

   GRAPHICS  4  (4 pont/byte, pontsorrend:  1 2 3 4 )
┌──────┬──────┬──────┬──────┬──────┬──────┬──────┬──────┐
│  B7  │  B6  │  B5  │  B4  │  B3  │  B2  │  B1  │  B0  │
├──────┼──────┼──────┼──────┼──────┼──────┼──────┼──────┤
│1.p. C0│2.p. C0│3.p. C0│4.p. C0│1.p. C1│2.p. C1│3.p. C1│4.p. C1│
└──────┴──────┴──────┴──────┴──────┴──────┴──────┴──────┘

   GRAPHICS 16  (2 pont/byte, pontsorrend:  1 2 )
┌──────┬──────┬──────┬──────┬──────┬──────┬──────┬──────┐
│  B7  │  B6  │  B5  │  B4  │  B3  │  B2  │  B1  │  B0  │
├──────┼──────┼──────┼──────┼──────┼──────┼──────┼──────┤
│1.p. I │2.p. I │1.p. G │2.p. G │1.p. R │2.p. R │1.p. B │2.p. B │
└──────┴──────┴──────┴──────┴──────┴──────┴──────┴──────┘
```

*Még azt figyelembe kell venni, hogy a sprite byte-onként kerül kirakásra, emiatt Graphics 4 módban **4-gyel**, Graphics 2-ben **8-cal**, míg Graphics 16-ban **2-vel** osztható szélességűnek kell lennie. Persze ettől még maga a látható sprite lehet keskenyebb, ilyenkor az első vagy az utolsó oszlopokban „üres” háttér színű pixeleket kell tenni. Az alábbi példa sprite **8 pixel** széles, de az utolsó oszlopa üres, így a sprite már csak **7 pixel** szélesnek fog látszani, pedig mi mind a 8x8 pixelt (8x**2 byte**-ot) kirakjuk:*



A **PUTSPRITE** egy szubrutin, amit a programunkból bárhonnan meghívhatunk, de a hívás előtt be kell állítanunk a **HL**, **DE** és **A** regiszterekbe a szubrutin paramétereit a következők szerint:

HL	a sprite adatainak kezdőcíme (sima címkét töltünk bele)
DE	a sprite kívánt pozíciója a képernyőn (**VRAM**-ban)
A	a sprite magassága pixelsorokban

1. Az elején meghívjuk a **VRAMON** szubrutint, ami a korábban ismertetett módon belapozza a **VRAM**-ot, hiszen másképp nem tudnánk a képernyőre írni.
2. Ezt követően beállítjuk a **PUTSPRITE** paramétereit a fenti táblázatnak megfelelő módon.
3. Majd meghívjuk a **PUTSPRITE** szubrutint, ami kiteszi a sprite-ot a **DE** regiszternek megfelelő képernyő-pozícióba, majd visszatér.
4. Ezt követően a **VRAMOFF** szubrutin segítségével visszaállítjuk a memória lapozást.
5. Egy **RET** utasítással pedig visszatérünk a hívóhoz, ami most a BASIC-et jelenti.

De mi történik a PUTSPRITE-on belül?

- Az elején **BC** regiszterbe **64**-et teszünk, később kiderül, hogy miért.
- Majd következik egy **LDI** utasítás, ami azt csinálja, hogy kiolvas egy byte ot a **HL** által mutatott címről *(ez ugye a sprite byte-jaira mutat)*, azt bemásolja a **DE** által mutatott címre *(ez pedig a megfelelő képernyőcímre mutat*), majd növeli mindkét regiszter értékét egyel, ami nekünk szuper, mert így **HL** már a következő sprite byte-ra mutat, **DE** pedig a következő képernyő byte-ra, ezért tudjuk is majd folytatni a kirakást. Mivel **16 pixel** széles a sprite-unk, az meg pont **4 byte**, így négyszer ismételjük ezt, **négy** **LDI** utasítást használva.
- A **négy** **LDI** utasítással kitettünk egy sort a sprite-ból, most a **DE** regisztert át kellene terelni a következő képernyősorba. Azt tudjuk, hogy egy képernyősor **64 byte**, így a következő sor **64 byte-tal** van „arrébb” a **VRAM**-ban, tehát a **DE**-hez **64**-et kellene hozzáadni, hogy a következő sorra mutasson. Mi ravasz módon be is tettünk a **BC** regiszterbe **64**-et a az elején, csakhogy, mivel minden **LDI** egyel növelte a **DE** regisztert *(és a **HL**-t is)*, így az viszont már **4 byte-tal** előrébb jár, szóval igazából már csak **60**-at kell hozzáadnunk, hogy a következő sorra mutasson. Milyen szerencse, hogy minden **LDI** a **BC**-t meg csökkenti egyel, és mivel négy **LDI** -t hajtottunk végre, így most a **BC**-ben már csak **60** van. *Mekkora szerencsénk van! ;-)* Na, akkor nincs is más dolgunk, mint **DE**-hez hozzáadni **BC**-t és máris a sprite következő sorának megfelelő képernyő-pozícióra fog mutatni. Igen ám, de a *Z80* proci tervezői olyan furfangosan állították össze az utasításokat, hogy csak **HL**, **IX** és **IY** regiszterekhez lehet egy másik regisztert hozzáadni, **DE**-hez nem. De annyi menekülőutat azért adtak, hogy van olyan lehetőségünk, hogy **DE**-t és **HL**-t felcseréljük az **EX DE,HL** utasítással. Ekkor ugye **HL**-be kerül **DE** értéke (és fordítva), így már egy **ADD HL,BC** utasítással jók is vagyunk. Ezután ügyesen visszacsináljuk a cserét egy ismételt **EX DE,HL** -el és **DE** már a következő sorra is mutat, miközben **HL** igazából

nem változott. *Hurrá!* Tudjuk, hogy minden **LDI** növelte **HL** értékét is, így az a soron következő sprite byte-ra mutat, és mivel a sprite byte-ok egymás után vannak ömlesztve a memóriában, így ezzel semmi több dolgunk nincs. *Nagyon jól nyomjuk! :)*

- Kiraktunk egy sort, **DE** a következő képernyősor megfelelő pozíciójára mutat, **HL** is rendben van, tehát jöhet is a következő sor kirakása. Ehhez előbb csökkentjük **A** regiszter értékét, amibe az elején a sprite sorainak számát tettük. Ha ez nem nulla, akkor még van következő kirakandó sora a sprite-nak, így visszaugrunk a ciklus elejére (**JP NZ,PUTSPRITE** – **J**um**P** if **N**on **Z**ero) és kezdjük elölről a folyamatot, amíg **A** regiszter nullára nem csökken, mert az azt jelenti, hogy kiraktuk az összes sort, azaz készen vagyunk.
- A végén jön még egy kecses **RET** *(RETURN)*, amivel visszatérünk a szubrutin hívásának helyére.

És akkor tisztázzuk, hogy miért jó ez, hogy fixen beraktunk ebbe a sprite-kirakóba annyi **LDI** *utasítást, ahány byte széles a sprite, amit kirakunk. Mert, hogy van egy olyan utasítás is, hogy* ***LDIR****, ami azt csinálja, hogy* ***BC*** *számú* ***LDI****-t hajt végre tök automatikusan (****LDI R****epeat –* ***LDI*** *ismétlés). Betöltjük* ***BC****-be, hogy hányszor szeretnénk ismételni és csak kiadjuk egyetlenegyszer, hogy* ***LDIR****, és készen vagyunk. Így* ***BC*** *regiszterbe, mint paraméterbe át tudnánk adni a sprite szélességét byte-okban és lenne egy teljesen általános sprite-kirakónk, ami tetszőleges szélességű sprite-ot ki tudna rakni a kód módosítása nélkül. Ez igaz, házi feladatnak tessék megcsinálni!*

De akkor mi a manóért nem így csináltuk?

Azért, mert egy ***LDIR*** ***20*** *CPU clock végrehajtású idejű minden ismétléskor és erre még rájön* ***1*** *CPU clock zárásként, míg egy* ***LDI*** *csak* ***16*** *CPU clock, tehát* ***20%-kal kevesebb idő alatt végez****, nekünk meg egy nem túl gyors TVC-nk van, így mindenhol érdemes gyorsítani, ahol csak lehet. Nem nagy dolog annyi sprite-kirakót írni, ahány fajta szélességű sprite-unk van. Elég rövidke kis szubrutin ez :)*

De nem csak sprite kirakásra jó ez a fajta képminta-kirakó, hanem akár pályák megjelenítésére is, úgynevezett **Tile Map** technikával, aminek a segítségével kis memóriafoglalás mellett tudunk sok pályát eltárolni, amik ráadásul relatív gyorsan kirakhatók / válthatók a képernyőn. Erről fog szólni a következő fejezet.

6. Tile map – a kicsi és gyors pályatérkép

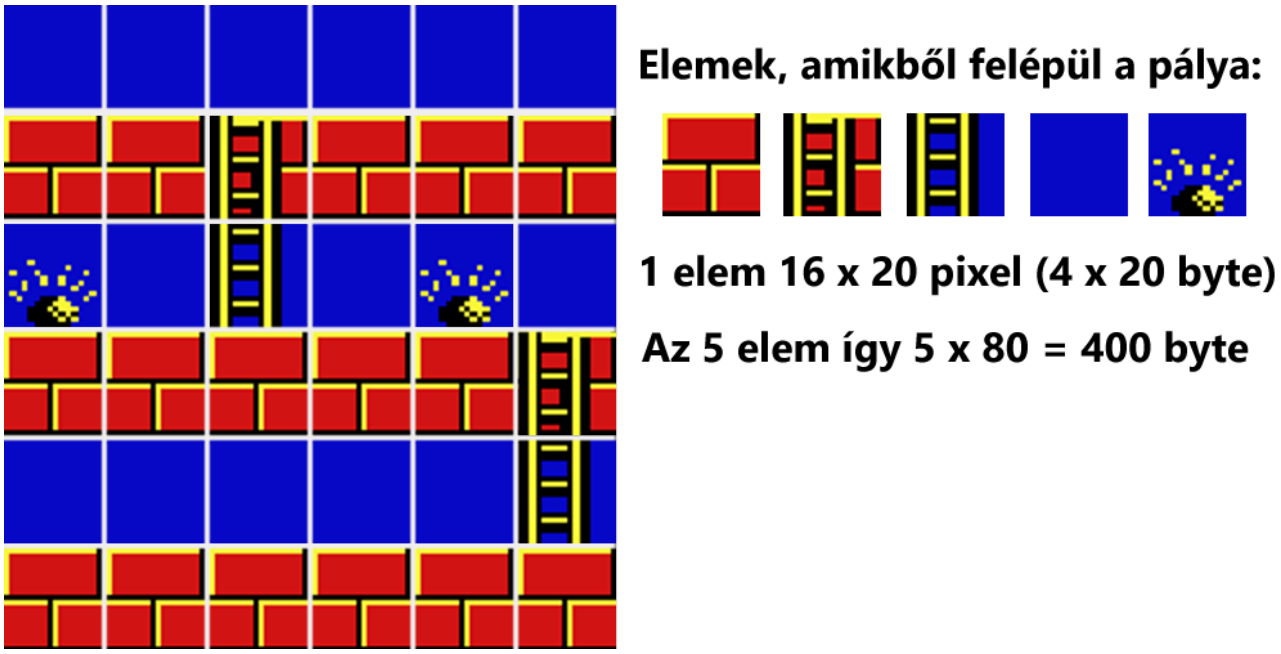
Tudjuk, hogy a videomemória képernyőn látható része **15 360 byte**, azaz egy teljes képernyős grafika **15 KByte** méretű *(lásd a legelső fejezetben)*. Amikor bekapcsolunk egy 64 KByte-os *TVC*-t, az induláskor kiírja, hogy **42 255 bytes** free *(2.2-es verzión 42 155)*, azaz hogy nem egész **42 Kbyte**-nyi szabad helyünk van, amivel gazdálkodhatunk. Azaz nincs 3 egész képernyőt betöltő grafikára sem helyünk és akkor még hol van maga a kód?!

De akkor, hogy lehet, hogy egyes játékoknak, mint a Kísértetkastély, a Ladder Man, a Spherical vagy az Expedíció van egy csomó pályája? Azok hol tárolódnak, honnan másolódnak ki a képernyőre?

Ez egy egyszerű technikával van megoldva, aminek **Tile map** a neve, ami lefordítva *Csempe-térkép*et jelent. Na ez jó bután hangzik magyarul, de mindjárt elmagyarázom és akkor értelmet nyer.

Ezeknél a játékoknál a pályák fel vannak osztva négyzetekre, mint a **csempe** a falon, és a grafikai elemek ilyen **csempékből** állnak, amikből egy csomó általában ismétlődik, így akár egészen kevés csempéből, azaz grafikai elemből fel lehet építeni egy képernyőnyi pályát.

Mutatom egy ábrán:

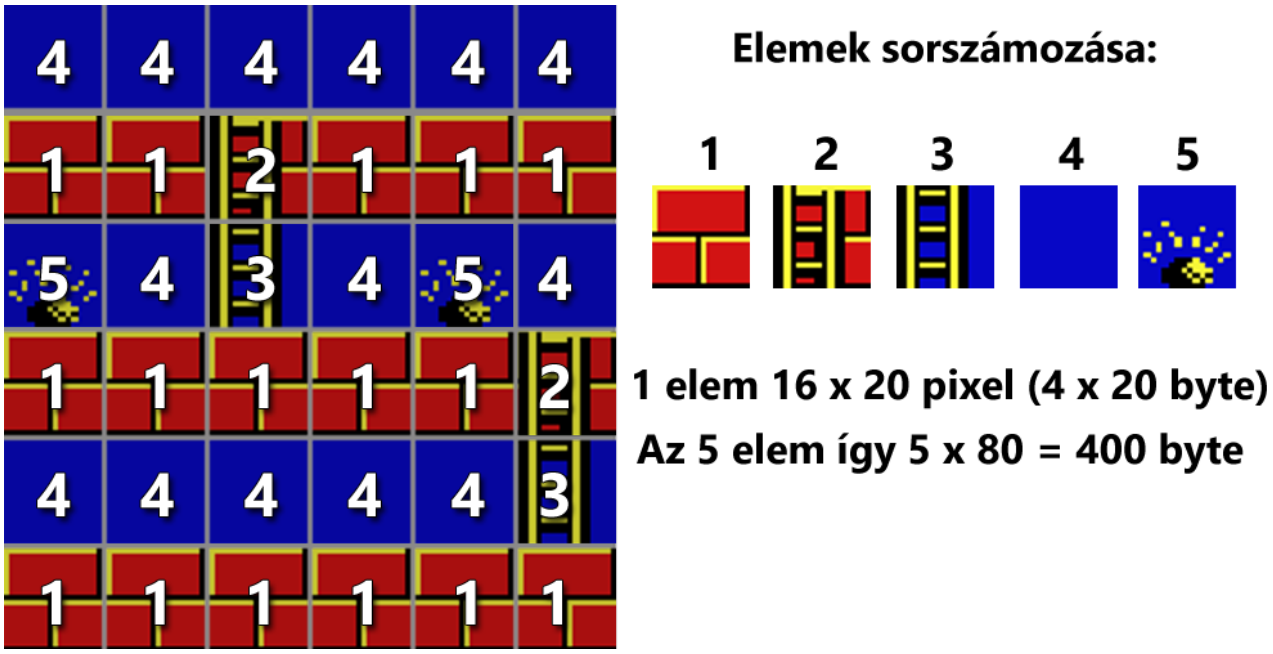


Ez az általam 1989-ben elkövetett *Ladder Man* játék egy részlete, illetve pályáinak építőelemei *(csempéi)*.

A bal oldali részlet jól szemlélteti, hogy épül fel egy pálya ebből a mindössze 5-féle *csempéből*. Az ábrán van a csempék között *„fuga”*, hogy a példa kedvéért jól elkülönüljenek egymástól, de valójában persze nincs közöttük semmilyen hézag, így összefüggő képet adnak.

Oké, már értjük a csempe részét, de hol a térkép?

Nézzük az előbbi példán, hogy néz ki a képernyő csempe-térképe:



Az ábrán, a bal oldali képernyőrészlet csempéin fehér számmal szerepel, hogy az adott képernyő pozícióra hányas sorszámú csempét kell kitenni a jobb oldalon látható számozás szerint.

Ez alapján térképnek azt a számsorozatot nevezzük, ami leírja, hogy a képernyőn melyik sorszámú csempe hová kerüljön. Tehát ez nem más, mint szimplán balról-jobbra, sorról-sorra haladva a csempéink sorszámai, szépen egymás után.

Ha csak ekkora lenne a képernyőnk, mint ami az ábrán látszik, tehát 6x6 csempe, akkor így nézne ki a csempe-térképe:
4, 4, 4, 4, 4, 4, 1, 1, 2, 1, 1, 1, 5, 4, 3, 4, 5, 4, 1, 1, 1, 1, 1, 2, 4, 4, 4, 4, 4, 3, 1, 1, 1, 1, 1, 1

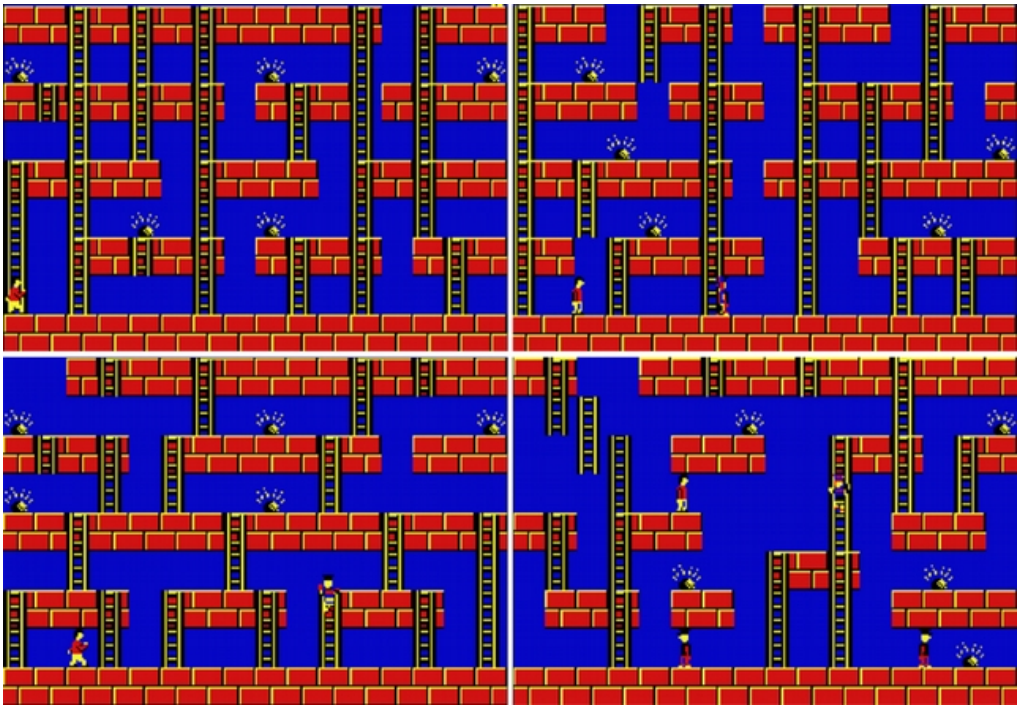
De akkor matekozzunk, hogy miként is tud ennyi pálya elférni abban a kevés rendelkezésre álló memóriában. Gondolom már sejted :)

Ha abból indulunk ki, hogy a pálya teljes képernyős, ami Graphics 4 módban **256x240** pixelt jelent, akkor a **16x20** pixel méretű csempékből vízszintesen **16** darabot tudunk kitenni és ilyen csempesorból függőlegesen **12** sor alkot egy teljes képernyőt. Azaz egy **16x12** = **192** számból álló *„térkép”* le tud írni egy teljes képernyőt. Tehát **192** byte-ból megvan egy egész pályánk. Na, hát ebből elég sok elfér a rendelkezésünkre álló **42 155** byte-on.

Ehhez persze még hozzájönnek a csempéink grafikai elemei. Egy ilyen elem **16x20** pixel méretű, azaz egy csempe **4x20** byte = **80** byte *(Graphics 4-ben 1 byte = 4 pixel)*.
Az 5 csempe tehát **5x80 = 400 byte**. Hát ez sem túl sok. Így már érthető, hogyan fér el 50 vagy akár 100 pálya is egy játékban.

A példában szereplő 5-féle csempéből számos pályát meg lehet építeni. A *Ladder Man* játékban egyébként **50 pálya** van, de csak mert úgy gondoltam, hogy az már elég sok, egyébként fért még volna akár több is.

Néhány a pályák közül (mindegyik az ismertetett 5-féle csempéből áll)



És mivel ilyen kevés helyet foglal egy csempe-térkép, így marad helyünk akár 5-nél több csempe-félét használni, hogy változatos legyen a játékunk megjelenése. A későbbi pályák így akár egészen eltérően nézhetnek ki, ahogy a *Kísértetkastély*ban is változik bizonyos pályákon kövezet mintázata és a színe is. Az *Expedíció* c. játékban pedig többféle háttérelem is megjelenik (oszlopok, szobrok, tárgyak). A csempeméret sincs kőbe vésve, használhatunk kisebb vagy nagyobb csempéket is, ahogy a játékunk grafikája kiadja, illetve megkívánja.

Ráadásul az így tárolt pályák viszonylag gyorsan kirakhatók. Nézzük csak meg a *Kísértetkastély*ban vagy az *Expedíció*ban, amikor a főhős kimegy a képernyőről, akkor egészen gyorsan megjelenik a következő pálya.

Bár nyilván sokan összerakták már fejben, de akiknek még nem teljesen tiszta, azok kedvéért nézzük meg, hogy kell egy ilyen pályát kirakni:

1. Belapozzuk a VRAM-ot és megcímezzük az elejét.
2. Egy másik mutatót az aktuális pályánk csempe-térképének elejére állítjuk.
3. Kiolvassuk egy sorszámot a csempetérképről.
4. A kiolvasott sorszám alapján megcímezzük a megfelelő csempénk sprite-ját.
5. Kirakjuk a képernyőre a sorszámnak megfelelő csempét *(a **PutSprite** rutin tökéletes erre)*
6. Tovább léptetjük a képernyőcímet egy csempe szélességgel.
7. Tovább lépünk a következő sorszámra a csempe-térképünkben.
8. Ha még nem raktunk ki egy teljes sort, akkor ismételünk a 3. ponttól, amíg nincs kész.
9. Tovább lépünk a VRAM-ban az egy csempesorral lentebbi címre.
10. Ha még nem értünk a képernyő vagy a pályánk aljára, akkor a 3 ponttól ismételve megkezdünk egy újabb sort kirakni.

Ennyi.

Ez volt az elmélet, de akkor lássuk, hogy néz ez ki assembly-ben.

A példaprogram az alábbi linkről tölthető le:
Tile Map Demo

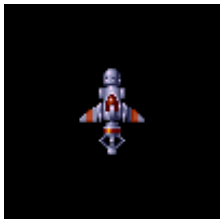
7. Sprite mozgatás

Már raktunk ki sprite-ot a képernyőre, de egy játékban alapvetően ezek a figurák - jó esetben - meg is mozdulnak, így aztán itt az ideje, hogy ennek is utána nézzünk.

Első körben a dolgok könnyebbik végét fogjuk meg, egyrészt mert a programozók lusták ;-), másrészt mert bizonyos esetekben ez a módszer nagyon is működik és ráadásul gyorsabb is, mint a későbbiekben majd ismertetésre kerülő, minden esetben használható megoldás.

Mire gondoltam, amikor a dolgok könnyebbik végéről beszéltem?
Eddig még csak olyan sprite kirakónk van, ami felülírja a hátteret. Láttuk, hogy ez Tile Map pályarajzolásnál teljesen jó, de olyan sprite-oknál is megfelelő, amik egyszínű háttér előtt mozognak. Ilyen lehetne például a *Kísértetkastély*ban az ellenfelek kirakása, de a villogás alapján valószínűleg azoknak a sprite-oknak is átlátszó a hátterük. De viszont simán lehet ilyen egy fekete űrbe mozgó űrhajó, ami mögött csak a nagy, egybefüggő feketeség a háttér.

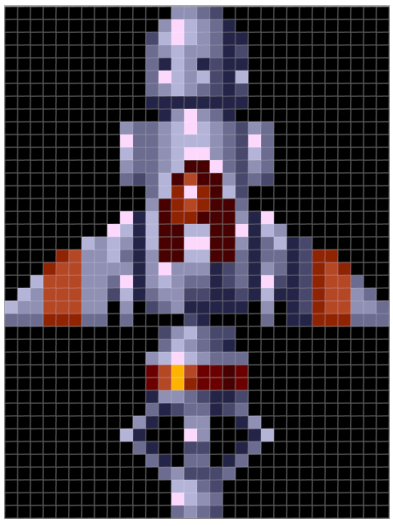
Például így



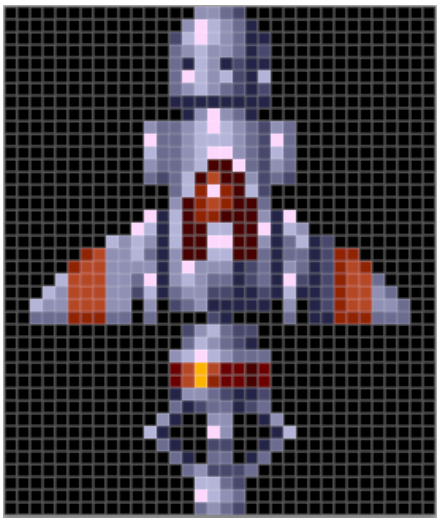
Na, de hogy tudunk ezzel trükközni? Úgy, hogy annyival nagyobbra vesszük az űrhajónk sprite-ját minden irányba, mint ahány pixelnyit egyszerre elmozdul. Azaz, ha mondjuk csak jobbra-balra lehet mozgatni és 2 pixelt mozdul el egyszerre, akkor mind a jobb, mint a bal szélén hagyunk 2-2 pixelnyi üres helyet a sprite-ban. Ha fel-le is mozog, akkor persze ott is annyi üres helyet hagyunk.

És miért jó ez? Mert amikor elmozdul, akkor saját magát írja felül és egyben törli a korábbi képernyő pozícióban, így nem kell a háttér elmentésével és visszaállításával foglalkoznunk, amivel sok időt nyerünk.

Ez az eredeti sprite



Ez pedig a 2-2 pixellel szélesebb



Akkor az elméleti résszel megvagyunk, nézzük meg, hogy fest ez a gyakorlatban.

A példaprogramban egy űrhajót tudunk jobbra-balra mozgatni a joystick-el, űrfekete háttér előtt, Graphics 16 módban.

*Ha a **PUT_SHIP_SPRITE** résznél kivesszük a HALT utasítás, akkor felgyorsul a történet :-)*

A példába egy univerzális sprite-kirakó került **PUT_SHIP_SPRITE** néven, ami a sprite szélességét és magasságát magából a sprite-ból veszi olyan módon, hogy a sprite adatok első byte-ja a sprite szélessége byte-okban, a második pedig a magassága pixel-sorokban. Az ezeket követő byte-ok pedig a szokásos sprite pixelek byte-onként, egymás után ömlesztve. Az általam készített **BMPToTVCSprite** segédprogram tud ilyen módon sprite-okat kimenteni, de ha a sprite címkéje után egy assembly DB sorba simán beírjuk kézzel a szélesség és magasság értékeket vesszővel elválasztva, az is ugyanolyan jó. Ezzel így egy kompakt kis sprite kirakót lehet megvalósítani.

*Ha a sprite sorok kirakását nem ciklusban végezzük, hanem egymás alá írt annyi **LDI** utasítással, ahány byte széles a sprite, az továbbra is gyorsabb. De ez meg így elegánsabb :-)*

*A következő forráskód egy az egyben beilleszthető a **TVC Studio**-ba és fordítható, futtatható, csak a **CAS fájl létrehozása** opciót kell bepipálni.*

```
                        ; 3. példa - sprite mozgatás egyszínű háttér előtt
                        ORG  6639                   ; A program kezdőcíme
                        ;{BASIC Header
                        DB   $0F,$0A,$0,$DD         ; "10 PRINT" - basic token
                        DB   ' USR'                 ; " USR"
                        DB   $96                    ; "("
                        DB   '7000'                 ; "7000"
                        DB   $95,$FF                ; ")"
                        DS   346,$0                 ; 346 db 0
                             ;}

                        JP   INIT                   ; <- ez lesz a 7000-es ($1B58) cím

                        ;{Konstansok
                        ; billentyűzet mátrix                                ;                   |
JOY1_STATE              EQU  2905                   ; bill. mátrix 8.sor
                        ; Joystick irány bitek a mátrixban
JOY_LEFT                EQU  64
JOY_RIGHT               EQU  32
                        ; rendszerváltozók
P_SAVE                  EQU  3                      ; memória lapozás memória tükre
MEM_PAGES_PORT          EQU  2                      ; memória lapozás beállítás portja
GRAPH_MODE              EQU  $0B13                  ; GRAPHICS rendszerváltozó
GRAPHICS_MODE_PORT      EQU  6                      ; 0-1. bitek: graphics mode
                        ; képernyővel kapcsolatos konstansok
VRAM_START              EQU  $8000                  ; a VRAM kezdőcíme
                        ;}


                        ;{Változók
SHIP_X                  DB   0                      ; űrhajó X byte-okban (0-63)
SHIP_VRAM_POS           DW   0                      ; űrhajó VIDEORAM cím
                        ;}

                        ;{Init - kezdőképernyő és változók alapbeállítása
INIT                    LD   A,(GRAPH_MODE)         ; grafikus mód kiolvasása a rendszerváltozóból
                        AND  255-2-1                ; töröljük az alsó 2 bitet
                        OR   2                      ; beállítjuk a grafikus módot; 2: Graphics 16
                        LD   (GRAPH_MODE),A         ; kiírjuk
                        OUT  (GRAPHICS_MODE_PORT),A ; majd kiküldjük a 6-os portra is
                        RST  $30                    ; $30-as rendszer rutin hívás
                        DB   $5                     ; Képernyő törlése paraméterrel
                        LD   A,$50                  ; A video memória belapozása a $8000-$0BFFF
                        LD   (P_SAVE),A             ; érték beírása a P_SAVE rendszerváltozóba
                        OUT  (MEM_PAGES_PORT),A     ; és kiküldése a portra
                        LD   A,32                   ; A = 32
                        LD   (SHIP_X),A             ; SHIP_X = 32 - ez lesz az alapértéke
                        LD   DE,VRAM_START+190*64+32 ; képernyő 190. sor, 32*2=64.pixel
                        LD   (SHIP_VRAM_POS),DE     ; beállítjuk SHIP_VRAM_POS változóba
                        JP   PUT_SHIP_SPRITE        ; az űrhajó kirakása a képernyőre
                        ;}
```

```
                ;{MAIN_LOOP - a program főciklusa
MAIN_LOOP       LD   A,(JOY1_STATE)          ; A = joystick állapot-byte-ja
                CP   JOY_RIGHT               ; jobbra irány lett nyomva?
                JP   Z,MOVE_RIGHT            ; ha igen, ugrás a jobbra mozgatásra
                CP   JOY_LEFT                ; balra irány lett nyomva?
                JP   Z,MOVE_LEFT             ; ha igen, ugrás a balra mozgatásra
                JP   MAIN_LOOP               ; vissza a főciklus elejére
                ;}

                ;{MOVE_RIGHT - az űrhajó mozgatása jobbra
MOVE_RIGHT      LD   A,(SHIP_X)              ; A = SHIP_X változó értéke
                CP   63-10                   ; A = 63-10 -> a sprite 10 byte széles
                JP   Z,MAIN_LOOP             ; ha igen, akkor nem tudunk tovább menni
                INC  A                       ; A = A + 1
                LD   (SHIP_X),A              ; az új értéket elmentjük a változóba
                LD   HL,(SHIP_VRAM_POS)      ; HL = SHIP_VRAM_POS
                INC  HL                      ; HL = HL + 1 -> 1 byte-al elmozgatjuk
                LD   (SHIP_VRAM_POS),HL      ; az új értéket elmentjük a változóba
                JP   PUT_SHIP_SPRITE
                ;}

                ;{MOVE_LEFT - az űrhajó mozgatása jobbra
MOVE_LEFT       LD   A,(SHIP_X)              ; A = SHIP_X változó értéke
                CP   0                       ; A = 0 -> a képernyő bal szélén vagyunk?
                JP   Z,MAIN_LOOP             ; ha igen, akkor nem tudunk tovább menni
                DEC  A                       ; A = A - 1
                LD   (SHIP_X),A              ; az új értéket elmentjük a változóba
                LD   HL,(SHIP_VRAM_POS)      ; HL = SHIP_VRAM_POS
                DEC  HL                      ; HL = HL - 1 -> 1 byte-al elmozgatjuk
                LD   (SHIP_VRAM_POS),HL      ; az új értéket elmentjük a változóba
                ;}

                ;{PUT_SHIP_SPRITE - űrhajó sprite kirakása
PUT_SHIP_SPRITE LD   HL,SHIP_SPRITE          ; HL = a sprite-unk memóriacíme
                INC  HL                      ; lépés a következő byte-ra
                LD   B,(HL)                  ; B = a sprite magassága pixelsorokban
                LD   IXL,B                   ; IXL = B -> azaz a sprite magassága
                INC  HL                      ; lépés a következő byte-ra
                LD   DE,(SHIP_VRAM_POS)      ; DE = a sprite címe a VRAM-ban
                HALT                         ; megvárunk egy cursor-megszakítást
PUT_SHIP_Y_ITER LD   BC,64                   ; BC = 64 -> ennyi byte széles a képernyő
                LD   A,(SHIP_SPRITE)         ; A = a sprite szélessége byte-okban
PUT_SHIP_X_ITER LDI                          ; egy byte sprite kirakása
                DEC  A                       ; A = A - 1
                JP   NZ,PUT_SHIP_X_ITER      ; ha A nem nulla, akkor folytatjuk
                EX   DE,HL                   ; DE <=> HL csere
                ADD  HL,BC                   ; HL = HL + BC
                EX   DE,HL                   ; DE <=> HL csere vissza
                DEC  IXL                     ; IXL = IXL - 1 -> ennyi sort kell még kirakni
                JP   NZ,PUT_SHIP_Y_ITER      ; ha IXL nem nulla, akkor még vannak sorok
                JP   MAIN_LOOP               ; készen vagyunk, ugrás vissza a főciklusra
                ;}

                ;{SHIP_SPRITE 20x42 pixel (10x42 byte)
SHIP_SPRITE     DB   10,42                   ;sprite width, height
                DB   0,0,0,0,1,2,0,0,0,0
                DB   0,0,0,0,87,43,0,0,0,0
                DB   0,0,0,0,87,43,0,0,0,0
                DB   0,0,0,1,191,63,2,0,0,0
                DB   0,0,0,1,191,63,2,0,0,0
                DB   0,0,0,1,191,63,2,0,0,0
                DB   0,0,0,1,191,63,2,0,0,0
                DB   0,0,0,1,191,63,2,0,0,0
                DB   0,0,0,1,191,63,2,0,0,0
                DB   0,0,0,1,191,63,2,0,0,0
                DB   0,0,0,1,42,0,2,0,0,0
                DB   0,0,0,87,63,63,43,0,0,0
                DB   0,0,0,87,63,63,43,0,0,0
                DB   0,0,0,87,127,151,43,0,0,0
                DB   0,0,0,85,127,151,2,0,0,0
                DB   0,0,0,1,251,195,2,0,0,0
                DB   0,0,0,1,251,195,2,0,0,0
                DB   0,0,0,87,243,195,43,0,0,0
                DB   0,0,1,191,243,195,63,2,0,0
                DB   0,0,87,63,243,195,63,9,0,0
                DB   0,1,238,63,63,63,63,12,2,0
                DB   0,1,238,63,163,3,63,12,2,0
                DB   0,87,204,63,63,63,63,12,3,0
                DB   0,87,204,63,163,3,63,12,43,0
                DB   0,87,204,63,63,63,63,12,43,0
                DB   0,87,204,63,163,3,63,12,43,0
                DB   0,3,3,3,2,1,3,3,3,0
                DB   0,0,0,0,23,43,0,0,0,0
                DB   0,0,0,1,63,63,2,0,0,0
                DB   0,0,0,1,191,63,2,0,0,0
                DB   0,0,0,1,183,63,2,0,0,0
                DB   0,0,0,1,191,63,2,0,0,0
```

```
DB    0,0,0,1,163,3,2,0,0,0
DB    0,0,0,87,63,63,43,0,0,0
DB    0,0,0,83,131,3,3,0,0,0
DB    0,0,0,1,191,63,2,0,0,0
DB    0,0,0,1,163,3,2,0,0,0
DB    0,0,0,0,23,43,0,0,0,0
DB    0,0,0,0,68,136,0,0,0,0
DB    0,0,0,0,204,204,0,0,0,0
DB    0,0,0,0,236,220,0,0,0,0
DB    0,0,0,0,84,168,0,0,0,0
;}

END             ;A program vége
```

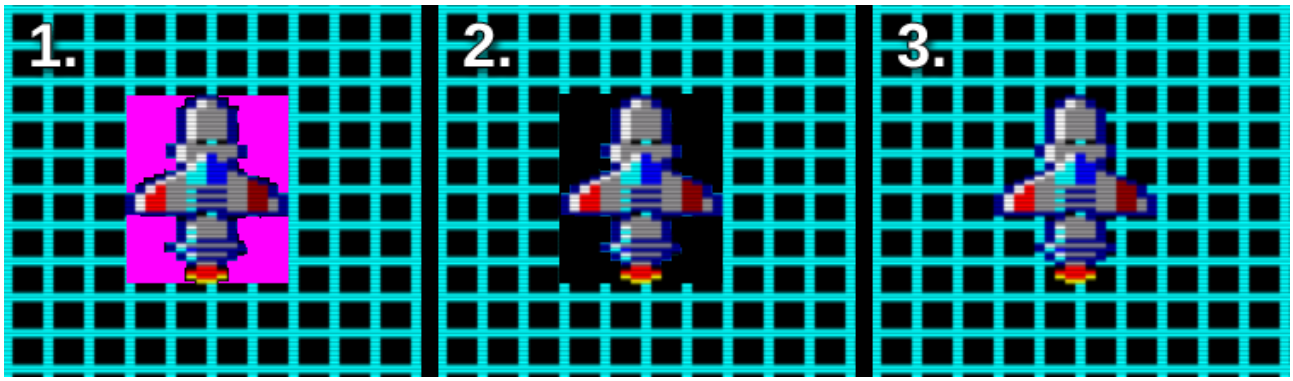
8. „Igazi” sprite kirakó

A reklámszlogen úgy szól, hogy a *Coca-Cola* az igazi, de nálunk most a **sprite** lesz az igazi :-)

De mitől igazi egy sprite?

Attól, hogy azok a pixelei, amik nem alkotják a sprite rajzolatát, átlátszóak, így bármilyen háttérre kirakhatóak, a háttér nem kívánt kitakarása / letakarása nélkül.

A következő ábra 1. képén az átlátszó pixelek lilával vannak jelölve. A 2. képen az előző fejezetben bemutatott, nem átlátszó sprite kirakás látható, míg a 3. képen az „igazi”, átlátszó sprite kirakás:

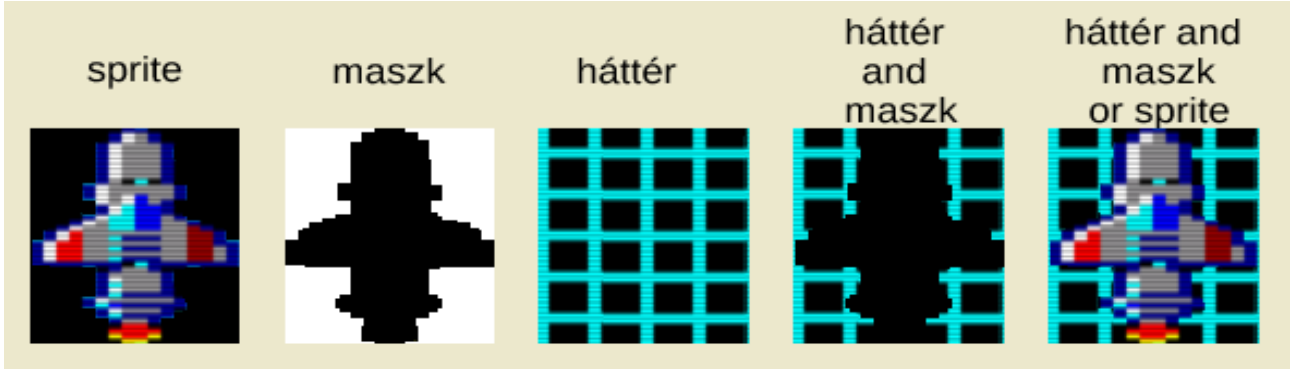


De hogyan tudjuk ezt az átlátszó kirakást megoldani?

Legkézenfekvőbb megoldás, hogy azokat a pixeleket, amik átlátszóak, egyszerűen nem rakjuk ki a képernyőre és készen is vagyunk. Ezzel csak annyi a baj, hogy ehhez a TVC összes grafikus módjában bit szinten kellene megvizsgáljuk a sprite-unk minden egyes byte-ját, majd kirakni azokat a biteket, amik látszanak és kihagyni, amik nem. Ez egy járható út, csak éppen ***lassú***.

Ennél gyorsabb, ha készítünk egy maszkot a sprite-hoz, amiben a sprite átlátszó pixeleinek az összes bit-je **1**-es, viszont azok a pixelei, ahol a látható pontjai vannak, ott az összes bit **0**. *Ez miért jó?* Mert ha ezt *logiai és*sel (**AND**) rákeverjük a képernyőre, az letöröl minden pixelt, ahol a sprite látható részei vannak, hiszen ott a maszkban mindenhol **0** van, viszont érintetlenül hagy minden pixelt, ahol a sprite átlátszó, hiszen ott a maszkban mindenhol **1** van. Erre pedig *logikai vagy*-gyal (**OR**) rá tudjuk keverni a sprite-ot, feltéve hogy az átlátszó helyeken **0** van a sprite-ban. ***Ezt bitenkénti művelet helyett byte-onként tudjuk elvégezni, ettől lesz gyorsabb.***

Íme a maszk és a maszkolt kirakás folyamata:



Ha mozgatni is szeretnénk a sprite-ot – márpedig többnyire ez így van –, akkor ennyi nem lesz elég még, mivel a sprite rajzolatát nem alkotó pixelek most nem jelennek meg, így azok nem is tudják törölni majd a sprite-unkat, ha elmozdítjuk, mint ahogy tették ezt az előző fejezetben, amikor egyszínű háttér előtt mozgattuk a sprite-ot. De hát az amúgy sem lenne jó most nekünk, hiszen nem egyszínű háttere szeretnénk a sprite-ot kitenni, ezért abba nem is szeretnénk beletörölni.

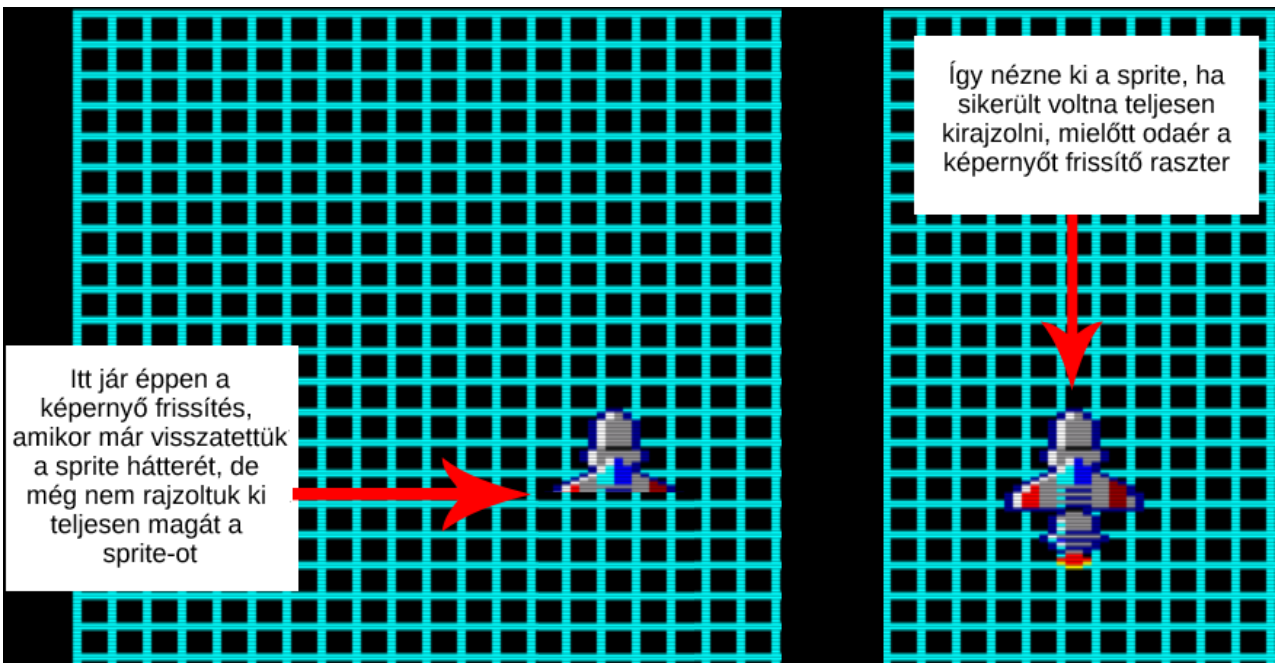
Akkor mi a megoldás a sprite mozgatásra?

1. Kirakás előtt el kell menteni a sprite mögötti hátteret egy erre kijelölt memóriaterületre.
2. **AND** művelettel letörölni a sprite helyét a maszkkal a képernyőn.
3. **OR** művelettel „rákeverni” a sprite-ot a maszkkal előtörölt képernyőre.
0. Ha elmozdítjuk a sprite-ot, akkor a korábbi pozícióba visszamásolni az elmentett hátterét az **1-3** lépések előtt.

Raszter-megszakítás átállítása

A fenti módszer bár gyorsabb, mint az előtte felvetett elmélet, amikor is pixelenként vizsgáljuk, hogy átlátszó-e a sprite adott képpontja vagy sem, de azért a háttér visszarakással, új háttér elmentéssel, sprite maszkolt kirakással együtt mégsem annyira gyors, hogy akár egy átlagos sprite is ne villanhatna be attól, hogy a képernyő kirajzolás éppen akkor ér oda, amikor még csak visszatettük a sprite hátterét, de még nem rajzoltuk ki teljesen a sprite-ot az új pozícióba. Szóval a képernyő alsó pár sorától eltekintve, jó eséllyel mindenhol máshol villogni fog a sprite kirakás. Minél nagyobb a sprite, annál nagyobb az a terület a képernyőn, ahol villogni fog. A képernyő alján azért kisebb a villogás esélye, mivel alapértelmezetten felül, a képernyő tetején kezdődik a képernyő frissítése, így a képernyő aljára ér oda a legkésőbb, azaz ott áll rendelkezésre a legtöbb idő a sprite kirajzolására, mielőtt a raszter odaérne.

Így néz ki, amikor a raszter „belevág” a sprite kirakásba, mielőtt a teljes sprite-ot kiraktuk volna:



Oké, és mit tudunk tenni a sprite kirakás villogása, a raszter „bevágása” ellen?

Át tudjuk állítani a *raszter-megszakítás (cursor-megszakítás)* kezdőcímét. Ha a kirakandó sprite alá, vagy még inkább a sprite kezdőpozíciójának címe *(VIDEORAM címe)* alá állítjuk pár sorral a raszter-megszakítás kezdetét, akkor csaknem egy teljes képernyő-frissítésnyi idő áll majd rendelkezésünkre, hogy visszategyük a sprite hátterét, elmentsük az új pozícióból a hátteret és kirakjuk a sprite-ot az új képernyő pozícióba. Ez egészen nagy sprite-ok esetén is elegendő idő arra, hogy villogás mentesen meg tudjuk oldani a sprite kirakását.
Ez az elmélet, de akkor lássuk a forráskódot!

A következő forráskód egy az egyben beilleszthető a ***TVC Studio****-ba és fordítható, futtatható, csak a* ***CAS fájl létrehozása*** *opciót kell bepipálni.*

```
                        ; átlátszó sprite mozgatás háttér előtt
                        ORG  6639               ; A program kezdőcíme
                        ;{BASIC Header
                        DB   $0F,$0A,$0,$DD     ; "10 PRINT" - basic token
                        DB   ' USR'             ; " USR"
                        DB   $96                ; "("
                        DB   '6659'             ; "6659"
                        DB   $95,$FF            ; ")"
                        DS   0,0,0,0,0          ; 5 db 0, hogy 6659-nél kezdődjön a kód
                        ;}

                        JP   INIT               ; <- ez lesz a 6659-es cím

                        ;{Konstansok
                        ; rendszerváltozók
P_SAVE                  EQU  3                  ; memória lapozás memória tükre
MEM_PAGES_PORT          EQU  2                  ; memória lapozás beállítás portja
GRAPH_MODE              EQU  $0B13              ; GRAPHICS rendszerváltozó
GRAPHICS_MODE_PORT      EQU  6                  ; 0-1. bitek: graphics mode
KEY_REPEAT_DELAY        EQU  2917               ; Auto repeat késleltetés 20 ms-es egységekben
JOY1_STATE              EQU  2905               ; bill. mátrix 8.sor, ahol a joystick kérdezhető
le
                        ; Joystick irány bitek a mátrixban
JOY_UP                  EQU  2
JOY_DOWN                EQU  4
JOY_LEFT                EQU  64
JOY_RIGHT               EQU  32
                        ; CRTC 6845
CRTC_REG_PORT           EQU  $70                ; CRTC regiszter kiválasztó port

CRTC_DATA_PORT          EQU  $71                ; CRTC regiszter adat port
CRTC_CUR_START_CHAR     EQU  10                 ; CRTC cursor-megszakítás kezdő karakter a sorban
CRTC_CUR_END_CHAR       EQU  11                 ; CRTC cursor-megszakítás befejező karakter
CRTC_R_CUR_POS_H        EQU  14                 ; CRTC cursor-megszakítás poz. reg.felső byte
CRTC_R_CUR_POS_L        EQU  15                 ; CRTC cursor-megszakítás poz. reg.alsó byte
                        ; képernyővel kapcsolatos konstansok
VRAM_START              EQU  $8000              ; a VIDEORAM kezdőcíme
SCREEN_WIDTH            EQU  64                 ; ennyi byte széles a képernyősor
SCREEN_HEIGHT           EQU  240                ; ennyi sor magas a képernyő
                        ; űrhajó sprite konstansok
SHIP_SPR_WIDTH          EQU  8                  ; űrhajó sprite szélessége byte-okban
SHIP_SPR_HEIGHT         EQU  42                 ; űrhajó sprite magassága pixelben
                        ;}


                        ;{Változók
SHIP_X                  DB   0                  ; űrhajó X (0-63)
SHIP_Y                  DB   0                  ; űrhajó Y (0-239)
SHIP_VRAM_POS           DW   0                  ; űrhajó VRAM cím
SHIP_BG_POS             DW   0                  ; az űrhajó háttere erről a címről lett elmentve
SAVE_SP                 DW   0                  ; StackPointer átmeneti mentésére
                        ;}

                        ;{Init - kezdőképernyő és változók alapbeállítása
INIT                    LD   A,(GRAPH_MODE)     ; A = grafikus mód (ami egyben a VOLUME port)
                        AND  255-2-1            ; töröljük az alsó 2 bitet
                        OR   2                  ; beállítjuk az alsó 2 biten a grafikus módot
                        LD   (GRAPH_MODE),A     ; kiírjuk a grafikus módot tartalmazó byte-ok
                        OUT  (GRAPHICS_MODE_PORT),A ; majd kiküldjük a 6-os portra is
                        LD   A,$50              ; A video memória belapozása a $8000-$0BFFF címre
                        LD   (P_SAVE),A         ; érték beírása a P_SAVE rendszerváltozóba
                        OUT  (MEM_PAGES_PORT),A ; és kiküldése a portra
                        LD   A,32               ; A = 32
                        LD   (SHIP_X),A         ; SHIP_X = 32 - ez lesz az alapértéke
                        LD   A,100              ; A = 100
```

```
                LD   (SHIP_Y),A          ; SHIP_Y = 100 - ez lesz az alapértéke
                LD   DE,VRAM_START+100*64+32 ; képernyő 100. sor, 32*2=64. pixel
                LD   (SHIP_VRAM_POS),DE  ; beállítjuk a változóba a kezdőértéknek
                LD   HL,KEY_REPEAT_DELAY ; keyboard delay címe
                LD   (HL),1              ; Auto repeat delay = 1 (azaz 20 ms)
                CALL FILL_BACKGROUND     ; kitesszük a háttér mintát
                CALL SAVE_SHIP_BG        ; elmentjük az űrhajó hátterét
                CALL DRAW_SPRITE         ; az űrhajó kirakása a képernyőre
                ;}

                ;{MAIN_LOOP - a program főciklusa
MAIN_LOOP       LD   A,(JOY1_STATE)      ; A = a joystick állapot-byte-ja
                CP   JOY_UP              ; fel irány lett nyomva?
                CALL Z,MOVE_UP           ; ha igen, ugrás a felfelé mozgatásra
                CP   JOY_DOWN            ; lefelé irány lett nyomva?
                CALL Z,MOVE_DOWN         ; ha igen, ugrás a lefelé mozgatásra
                CP   JOY_RIGHT           ; jobbra irány lett nyomva?
                CALL Z,MOVE_RIGHT        ; ha igen, ugrás a jobbra mozgatásra
                CP   JOY_LEFT            ; balra irány lett nyomva?
                CALL Z,MOVE_LEFT         ; ha igen, ugrás a balra mozgatásra
                JP   MAIN_LOOP           ; ugrás vissza a főciklus elejére
                ;}

                ;{MOVE_UP - az űrhajó mozgatása felelé
MOVE_UP         LD   A,(SHIP_Y)          ; A = SHIP_Y változó értéke
                CP   0                   ; űrhajó a képernyő tetején van?
                RET  Z                   ; ha igen, akkor nem tudunk tovább menni jobbra
                DEC  A                   ;
                DEC  A                   ;
                DEC  A                   ;
                DEC  A                   ; A = A - 4
                LD   (SHIP_Y),A          ; az új értéket elmentjük a változóba
                LD   HL,(SHIP_VRAM_POS)  ; HL = SHIP_VRAM_POS
                LD   BC,4*SCREEN_WIDTH   ; BC = négy képernyő sornyi byte
                OR   A                   ; Carry törlése a kivonáshoz
                SBC  HL,BC               ; HL = egy sorral fentebbi pozíció
                LD   (SHIP_VRAM_POS),HL  ; az új értéket elmentjük a változóba
                CALL DRAW_SHIP           ; kitesszük az űrhajót háttérkezeléssel
                RET
                ;}

                ;{MOVE_DOWN - az űrhajó mozgatása lefelé
MOVE_DOWN       LD   A,(SHIP_Y)          ; A = SHIP_Y változó értéke

                CP   SCREEN_HEIGHT-SHIP_SPR_HEIGHT ; kint van a képernyőről?
                RET  NC                  ; ha igen, akkor nem tudunk tovább menni
                INC  A
                INC  A
                INC  A
                INC  A                   ; A = A + 4
                LD   (SHIP_Y),A          ; az új értéket elmentjük a változóba
                LD   HL,(SHIP_VRAM_POS)  ; HL = SHIP_VRAM_POS
                LD   BC,4*SCREEN_WIDTH   ; BC = négy képernyő sornyi byte
                ADD  HL,BC               ; HL = egy sorral lentebbi pozíció
                LD   (SHIP_VRAM_POS),HL  ; az új értéket elmentjük a változóba
                CALL DRAW_SHIP           ; kitesszük az űrhajót háttérkezeléssel
                RET
                ;}

                ;{MOVE_RIGHT - az űrhajó mozgatása jobbra
MOVE_RIGHT      LD   A,(SHIP_X)          ; A = SHIP_X változó értéke
                CP   SCREEN_WIDTH-SHIP_SPR_WIDTH ; űrhajó a képernyő jobb szélén van?
                RET  Z                   ; ha igen, akkor nem tudunk tovább menni jobbra
                INC  A                   ; A = A + 1
                LD   (SHIP_X),A          ; az új értéket elmentjük a változóba
                LD   HL,(SHIP_VRAM_POS)  ; HL = SHIP_VRAM_POS
                INC  HL                  ; HL = HL + 1->1 byte-al elmozgatjuk, ami 2 pixel
                LD   (SHIP_VRAM_POS),HL  ; az új értéket elmentjük a változóba
                CALL DRAW_SHIP           ; kitesszük az űrhajót háttérkezeléssel
                RET
                ;}

                ;{MOVE_LEFT - az űrhajó mozgatása balra
MOVE_LEFT       LD   A,(SHIP_X)          ; A = SHIP_X változó értéke
                CP   0                   ; A = 0 -> a képernyő bal szélén vagyunk?
                RET  Z                   ; ha igen, akkor nem tudunk tovább menni balra
                DEC  A                   ; A = A - 1
                LD   (SHIP_X),A          ; az új értéket elmentjük a változóba
                LD   HL,(SHIP_VRAM_POS)  ; HL = SHIP_VRAM_POS
                DEC  HL                  ; HL = HL-1 -> 1 byte-al elmozgatjuk, ami 2 pixel
                LD   (SHIP_VRAM_POS),HL  ; az új értéket elmentjük a változóba
                CALL DRAW_SHIP           ; kitesszük az űrhajót háttérkezeléssel
                RET
                ;}

                ;{DRAW_SHIP - űrhajó kirajzolása háttér kezeléssel
DRAW_SHIP       CALL SET_CURSOR_IT       ; raszter-megszakítás pozíció beállítása
```

```
                    HALT                        ; várunk egy raszter-megszakítást

                    CALL RESTORE_SHIP_BG        ; visszarakjuk az űrhajó hátterét
                    CALL SAVE_SHIP_BG           ; elmentjük az új hátteret
                    CALL DRAW_SPRITE            ; kirakjuk az űrhajót
                    RET                         ; kész, vissza a hívóhoz
                    ;}

                    ;{DRAW_SPRITE
DRAW_SPRITE         LD   IX,SHIP_SPRITE
                    LD   DE,SHIP_MASK
                    LD   HL,(SHIP_VRAM_POS)
                    DI                          ; megszakítások tiltása, mivel SP-t használjuk
                    LD   (SAVE_SP),SP           ; SP regisztert elmentjük, mert használni fogjuk
                    LD   SP,SCREEN_WIDTH-SHIP_SPR_WIDTH ; SP=SCREEN_WIDTH - a sprite szélesség
                    LD   C,SHIP_SPR_HEIGHT      ; C  = sprite magassága pixelben
DRAW_SPR_ITER       LD   B,SHIP_SPR_WIDTH       ; B  = egy sornyi byte
DRAW_SPR_LINE       LD   A,(DE)                 ; A  = sprite maszk aktuális byte-ja
                    AND  (HL)                   ; A  = "maszk byte" AND "képernyő byte"
                    OR   (IX+0)                 ; A  = A OR "sprite byte"
                    LD   (HL),A                 ; kitesszük a képernyőre
                    INC  DE                     ; a következő sprite mask byte-ra léptetjük
                    INC  HL                     ; a következő képernyő byte-ra léptetjük
                    INC  IX                     ; a következő sprite byte-ra léptetjük
                    DJNZ DRAW_SPR_LINE          ; ha még nincs kész az egész sor, ugrás vissza
                    ADD  HL,SP                  ; HL = a következő sor címe a VIDEORAM-ban
                    DEC  C                      ; sprite magasság ciklusszámláló csökkentése
                    JP   NZ,DRAW_SPR_ITER       ; ha még kész az összes sor, ugrás
                    LD   SP,(SAVE_SP)           ; SP regiszter visszaállítása az elmentett értékre
                    EI                          ; megszakítások engedélyezése
                    RET
                    ;}

                    ;{SAVE_SHIP_BG - sprite mögötti képernyőterület elmentése
SAVE_SHIP_BG        LD   HL,(SHIP_VRAM_POS)     ; HL = innen kell menteni a sprite hátterét
                    LD   (SHIP_BG_POS),HL       ; eltároljuk honnan lett a háttér lementve
                    LD   DE,SHIP_BG_PUFFER      ; DE = sprite háttér puffer
                    LD   A,SHIP_SPR_HEIGHT      ; A  = sprite magassága
SAVE_SHIP_BG_ITER   LD   BC,SCREEN_WIDTH        ; BC = SCREEN_WIDTH
                    LDI                         ; annyi LDI kell, ahány byte széles a sprite
                    LDI                         ; az LDIR 20 clocks az LDI vszint csak 16,
                    LDI                         ; így sokat lehet nyerni az ismételt LDI-kel
                    LDI
                    LDI
                    LDI
                    LDI
                    LDI
                    LDI
                    LDI
                    ADD  HL,BC                  ; HL = HL + (64 - "kirakott byte-ok")
                    DEC  A                      ; csökkentjük a kiirandó sorok számát
                    JP   NZ,SAVE_SHIP_BG_ITER   ; ha nincs kész az összes sort, vissza az elejére
                    RET                         ; vissza a hívóhoz
                    ;}

                    ;{RESTORE_SHIP_BG - sprite elmentett hátterének visszarakás
RESTORE_SHIP_BG     LD   HL,SHIP_BG_PUFFER      ; HL = elmentett háttér puffer címe
                    LD   DE,(SHIP_BG_POS)       ; DE = ahová vissza kell tenn a sprite hátterét
                    LD   A,SHIP_SPR_HEIGHT      ; A  = a sprite magassága pixelsorokban
RESTORE_BG_ITER     LD   BC,SCREEN_WIDTH        ; BC = SCREEN_WIDTH (64 byte egy képernyősor)
                    LDI                         ; annyi LDI kell, ahány byte széles a sprite
                    LDI                         ; az LDIR végrehajtása 20 clocks, az LDI csak 16,
                    LDI                         ; így sokat lehet nyerni az ismételt LDI-kel
                    LDI
                    LDI
                    LDI
                    LDI
                    LDI
                    LDI
                    LDI
                    EX   DE,HL                  ; HL <=> DE (csere)
                    ADD  HL,BC                  ; HL = HL + (64 - kirakott byte-ok)
                    EX   DE,HL                  ; DE <=> HL
                    DEC  A                      ; csökkentjük a kiirandó sorok számát
                    JP   NZ,RESTORE_BG_ITER     ; ha még nincs kész, vissza az elejére
                    RET
                    ;}

                    ;{SET_CURSOR_IT - képernyő cursor-megszakítás átállítása
SET_CURSOR_IT       LD   A,(SHIP_Y)             ; A = SHIP_Y
                    ADD  A,10                   ; pár sort még hozzáadunk
                    LD   L,A                    ; L = SHIP_Y + pár sor
                    AND  3                      ; 1-2. bit kivételével mindet töröljük
                    LD   E,A                    ; E regiszterbe elmenjük
                    LD   A,L                    ; A = (SHIP_Y+pár sor)
                    SUB  E                      ; A = (SHIP_Y+pár sor)-((SHIP_Y + pár sor) mod 4)
                    LD   L,A                    ; L = (SHIP_Y+pár sor)-((SHIP_Y + pár sor) mod 4)
```

```
                        LD   H,0                ; H = 0
                        ADD  HL,HL              ; HL = HL * 2
                        ADD  HL,HL              ; HL = HL * 4
                        ADD  HL,HL              ; HL = HL * 8
                        ADD  HL,HL              ; HL = HL * 16 -> HL=16 * HL
                        INC  HL                 ; 1-el növeljük a HL-t
                        ; cursor-megszakítás pozíciójának beállítása
                        DI                      ; letiltjuk a megszakításokat
                        LD   A,CRTC_R_CUR_POS_H
                        OUT  (CRTC_REG_PORT),A  ; cursor-megszak.poz.felső byte reg.
                        LD   A,H
                        OUT  (CRTC_DATA_PORT),A ; kiszámolt felső byte kiküldése
                        LD   A,CRTC_R_CUR_POS_L
                        OUT  (CRTC_REG_PORT),A  ; cursor-megszak.poz.alsó byte reg.
                        LD   A,L
                        OUT  (CRTC_REG_PORT),A  ; kiszámolt alsó byte kiküldése
                        ; karaktersorban a kezdő- és befejező TV sor beállítása
                        LD   A,CRTC_CUR_START_CHAR
                        OUT  (CRTC_REG_PORT),A  ; kezdő TV sort beállító regiszerének kiválasztása
                        LD   A,E                ; A = ((SHIP_Y+SPR_HEIGHT) / 4) művelet maradéka
                        OUT  (CRTC_DATA_PORT),A ; kiküldjük a portra
                        LD   A,CRTC_CUR_END_CHAR
                        OUT  (CRTC_REG_PORT),A  ; befejező TV sort beállító reg. kiválasztása
                        LD   A,E                ; A = ((SHIP_Y+SPR_HEIGHT) / 4) művelet maradéka
                        OUT  (CRTC_DATA_PORT),A ; ide is kiküldjük ugyanazt
                        EI                      ; engedélyezzük amegszakításokat
                        RET
                        ;}

                        ;{FILL_BACKGROUND - négyzetrács háttér kirakása
FILL_BACKGROUND         LD   DE,VRAM_START
                        LD   B,30               ; B = ennyi sornyi hátteret rakunk ki
FILL_BG_LINES           LD   C,32               ; C = ennyi háttér sprite-ot rakunk ki egy sorba
FILL_BG_ROW             PUSH BC                 ; BC-t eltesszük a stack-re
                        PUSH DE                 ; DE-t eltesszük a stack-re
                        LD   HL,BG_TILE         ; HL = a 32x64 pixeles űr textura
                        LD   B,8                ; B = a sprite magassága pixelsorokban
                        LD   IXL,B              ; IXL = B -> azaz a sprite magassága
PUT_BG_Y_ITER           LD   BC,64              ; BC = 64 -> ennyi byte széles a képernyő
                        LD   A,2                ; A = a sprite szélessége byte-okban
PUT_BG_X_ITER           LDI                     ; 1 byte kirakása
                        DEC  A                  ; A = A - 1
                        JP   NZ,PUT_BG_X_ITER   ; ha A nem nulla, akkor folytatjuk a ciklust
                        EX   DE,HL              ; DE <=> HL csere
                        ADD  HL,BC              ; HL = HL + BC
                        EX   DE,HL              ; DE <=> HL csere vissza
                        DEC  IXL                ; IXL = IXL - 1 -> ennyi sort kell még kirakni
                        JP   NZ,PUT_BG_Y_ITER   ; ha IXL még nem nulla, grás a ciklus elejére
                         POP  DE                ; DE-t visszaállítjuk a stack-ről
                        INC  DE                 ; 2 byte széles egy háttér csempe, ezért
                        INC  DE                 ; DE = DE + 2 -> a következő csempe címe
                        POP  BC                 ; BC-t visszaállítjuk a stack-ről
                        DEC  C                  ; csökkentjük az egy sorba kirakandó háttércsempét
                        JP   NZ,FILL_BG_ROW     ; ha nem nulla, folytatjuk a kirakást
                        LD   A,B                ; A = B -> elmentjük B regisztert
                        EX   DE,HL              ;
                        LD   BC,7*SCREEN_WIDTH  ; (csempemagasság - 1) * képernyőszélesség
                        ADD  HL,BC              ; ennyit adunk DE-hez, hogy a
                        EX   DE,HL              ; következő csempesor címére mutasson
                        LD   B,A                ; B értéke vissza A-ból
                        DEC  B                  ; csökkentjük a kirakandó sorok számát
                        JP   NZ,FILL_BG_LINES   ; ha nem nulla, akkor folytatjuk a kirakást
                        RET                     ; vissza a hívóhoz
                        ;}

                        ;{SHIP_SPRITE és MASK - 16x42 pixel (8x42 byte)
SHIP_SPRITE             DB   0,0,0,1,2,0,0,0
                        DB   0,0,0,87,43,0,0,0
                        DB   0,0,0,87,43,0,0,0
                        DB   0,0,1,191,63,2,0,0
                        DB   0,0,1,191,63,2,0,0
                        DB   0,0,1,191,63,2,0,0
                        DB   0,0,1,191,63,2,0,0
                        DB   0,0,1,191,63,2,0,0
                        DB   0,0,1,191,63,2,0,0
                        DB   0,0,1,191,63,2,0,0
                        DB   0,0,1,42,0,2,0,0
                        DB   0,0,87,63,63,43,0,0
                        DB   0,0,87,63,63,43,0,0
                        DB   0,0,87,127,151,43,0,0
                        DB   0,0,85,127,151,2,0,0
                        DB   0,0,1,251,195,2,0,0
                        DB   0,0,1,251,195,2,0,0
                        DB   0,0,87,243,195,43,0,0
                        DB   0,1,191,243,195,63,2,0
                        DB   0,87,63,243,195,63,9,0
                        DB   1,238,63,63,63,63,12,2
```

```
                    DB    1,238,63,163,3,63,12,2
                    DB    87,204,63,63,63,63,12,3
                    DB    87,204,63,163,3,63,12,43
                    DB    87,204,63,63,63,63,12,43
                    DB    87,204,63,163,3,63,12,43
                    DB    3,3,3,2,1,3,3,3
                    DB    0,0,0,23,43,0,0,0
                    DB    0,0,1,63,63,2,0,0
                    DB    0,0,1,191,63,2,0,0
                    DB    0,0,1,183,63,2,0,0
                    DB    0,0,1,191,63,2,0,0
                    DB    0,0,1,163,3,2,0,0
                    DB    0,0,87,63,63,43,0,0
                    DB    0,0,83,131,3,3,0,0
                    DB    0,0,1,191,63,2,0,0
                    DB    0,0,1,163,3,2,0,0
                    DB    0,0,0,23,43,0,0,0
                    DB    0,0,0,68,136,0,0,0
                    DB    0,0,0,204,204,0,0,0
                    DB    0,0,0,236,220,0,0,0
                    DB    0,0,0,84,168,0,0,0

SHIP_MASK           DB    255,255,255,170,85,255,255,255
                    DB    255,255,255,0,0,255,255,255
                    DB    255,255,255,0,0,255,255,255
                    DB    255,255,170,0,0,85,255,255
                    DB    255,255,170,0,0,85,255,255
                    DB    255,255,170,0,0,85,255,255
                    DB    255,255,170,0,0,85,255,255
                    DB    255,255,170,0,0,85,255,255
                    DB    255,255,170,0,0,85,255,255
                    DB    255,255,170,0,0,85,255,255
                    DB    255,255,170,85,255,85,255,255
                    DB    255,255,0,0,0,0,255,255
                    DB    255,255,0,0,0,0,255,255
                    DB    255,255,0,0,0,0,255,255
                    DB    255,255,170,0,0,85,255,255
                    DB    255,255,170,0,0,85,255,255
                    DB    255,255,170,0,0,85,255,255
                    DB    255,255,0,0,0,0,255,255
                    DB    255,170,0,0,0,0,85,255
                    DB    255,0,0,0,0,0,0,255
                    DB    170,0,0,0,0,0,0,85
                    DB    170,0,0,0,0,0,0,85
                    DB    0,0,0,0,0,0,0,0
                    DB    0,0,0,0,0,0,0,0
                    DB    0,0,0,0,0,0,0,0
                    DB    0,0,0,0,0,0,0,0
                    DB    0,0,0,85,170,0,0,0
                    DB    255,255,255,0,0,255,255,255
                    DB    255,255,170,0,0,85,255,255
                    DB    255,255,170,0,0,85,255,255
                    DB    255,255,170,0,0,85,255,255
                    DB    255,255,170,0,0,85,255,255
                    DB    255,255,170,0,0,85,255,255
                    DB    255,255,0,0,0,0,255,255
                    DB    255,255,0,0,0,0,255,255
                    DB    255,255,170,0,0,85,255,255
                    DB    255,255,170,0,0,85,255,255
                    DB    255,255,255,0,0,255,255,255
                    DB    255,255,255,170,85,255,255,255
                    DB    255,255,255,0,0,255,255,255
                    DB    255,255,255,0,0,255,255,255
                    DB    255,255,255,170,85,255,255,255
                    ;}

                    ;{BG_TILE - háttér csempe - 4x8 pixel (2x8 byte)
BG_TILE             DB    243,243
                    DB    243,243
                    DB    162,0
                    DB    162,0
                    DB    162,0
                    DB    162,0
                    DB    162,0
                    DB    162,0
                    ;}

                    ;{SHIP_BG_PUFFER - űrhajó háttér elmentésének puffere
SHIP_BG_PUFFER      DS    SHIP_SPR_WIDTH*SHIP_SPR_HEIGHT,0 ; puffer a sprite hátterének
                    ;}

END             ;A program vége
```

9. Tippek és trükkök sprite kirakáshoz

Hamarosan jön ez is.

10. Hangoskodjunk

A TVC hangképzési lehetőségei nagyon egyszerűek. Négy dolgot tudunk a hangkeltéssel kapcsolatosan vezérelni:

1. szóljon-e hang vagy ne (**SOUND**)
2. milyen hangmagasságú hang szólaljon meg (**PITCH 0 - 4095**)
3. mekkora hangereje legyen a hangnak (**VOLUME 0 - 15**)
4. mennyi ideig szóljon a hang (ez BASIC-ben a **DURATION 1 - nn**)

A 4. lehetőséget, a hangkitartás időtartamát, a BASIC-ben DURATION-ként ismert paramétert, gépi kódban magunknak kell vezérelni, a hangkeltés egyszerűsége miatt.

Alapvetően minden hanggal kapcsolatos lehetőséget portokon keresztül érünk el. A **4**-es, **5**-ös és **6**-os port megfelelő bitjei végzik a hangkeltést. De nézzük sorban a felsorolt 4 lehetőséget, ami már tudjuk, hogy valójában csak három, mivel a negyediket magunknak kell megvalósítanunk.

1. Hangjel

Az **5-ös port 4. bitjén** (*Hangjel*) keresztül tudjuk vezérelni, hogy szóljon-e hang, vagy sem. Ha szeretnénk egy hangot megszólaltatni, akkor fontos, hogy ez a bit **1-re** legyen állítva, azaz engedélyezve legyen a *hangjel*. Viszont, ha el akarjuk hallgattatni az éppen szóló hangot, akár csak egy rövidebb szünet idejére, akkor ehhez elég ezt a bitet **0-ra** állítani. Egyszerű és gyors módszer.

PORT05H (5) iras

B7	B6	B5	B4	B3	B2	B1	B0
+	+	Hang IT	Hangjel	PITCH felső 4 bit			
		0: tilt	0: tilt				
		1: eng	1: eng				

```
0B12H  2834  PORT05  DEFS  1  ; Az 5-ös I/O port tükörképe
```

Hang IT

A 4. bit mellett ott van mindjárt az **5.** is, ami a *hang megszakítás* (*hang interrupt* - *Hang IT*) engedélyezését és tiltását végzi. Ezt ajánlott tiltani. *Miért?* Azért, mert ezt a BASIC használja a 4. funkcióra a hanggal kapcsolatban, azaz arra, hogy a megszólaltatott hang kitartásának idejét mérje a *hang megszakítás* (*Hang IT*) segítségével. A megszakítás minden 20 ezredmásodpercben *(20 ms)* bekövetkezik - *pont ennyi 1 DURATION, nem véletlenül :)* -, így nagyszerűen lehet vele időt mérni. De ez egyben azt is jelenti, hogy két megszakításunk lesz, egyik a cursor- vagy raszter-megszakítás, ami a képernyő frissítésének a kezdetét jelzi, a másik pedig a hang megszakítás. Na most, ha a sprite-kirakónkban HALT utasítással várunk egy megszakításra, akkor az egyszer a cursor-megszakítás lesz, amire szükségünk is van, máskor meg a hang megszakítás, amire meg igazából semmi szükségünk nincs. **Így célszerű a Hang IT-t mindig letiltani, azaz az 5. bitre 0-át írni.**

Hogy néz ez ki a gyakorlatban?

Mivel az **5-ös** port nem csak ezt a két bitet tartalmazza, ezt érdemes figyelembe vennünk, ha adatot küldünk ki rá. A 6. és 7. bitek a kazetta motorvezérlését végzik, az alsó négy bit pedig a hangmagasság (PITCH) felső 4 bitjét adják, amit nem mindig érdemes felülírnunk, mert később még jól jöhet, hogy az utoljára kiadott hang értéke van benne. Így, ha a *Hangjel* vagy a *Hang IT* biteket szeretnénk beállítani, akkor előbb érdemes az **5-ös** port aktuális értékét kiolvasni az azt tartalmazó rendszerváltozóból, ami a decimális **2834**-es memóriacímen található és gyakorlatilag az ***5-ös port tükörképe***. Aztán ebből a kiolvasott byte-ból annyit bitet megváltoztatni, amennyire szükségünk van.

```
LD   A,(2834)  ; A regiszterbe betöltjük az 5-ös port tükrét
AND  128+64    ; a felső 2. bit kivételével a többit töröljük
OR   16        ; a 4. bitbe 1-et írva engedélyezzük a Hangjelet
LD   (2834),A  ; visszaírjuk a módosult értéket a rendszerváltozóba
OUT  (5),A     ; és kiküldjük az értéket az 5-ös porta
```

Persze ennyitől még semmi nem történik, ez csak a hang kiadásának előkészítése. Jöhet a második lépés, ahol a hangmagasságok állítjuk be.

2. Hangmagasság (PITCH)

A hangmagasságot a **4-es** és az **5-ös** porta kiküldött PITCH értékkel tudjuk beállítani. Ez ráadásul azonnal meg is szólal, ha a *Hangjel* engedélyezve van.

Tehát a **PITCH** értékét két porton keresztül tudjuk megadni, mivel egy portra, csak 1 byte, azaz 8 bit küldhető ki, ami 0 - 255 közötti szám ábrázolását teszi csak lehetővé, de a TVC **0 - 4095** közötti értékeket tud megszólaltatni. Ezt **12 biten** tudjuk ábrázolni, ezért van szükség a **4-es port 8 bitje** mellett az **5-ös port alsó 4 bitjére** is.

Például a 4000-es PITCH értéket úgy tudjuk kiküldeni, hogy a 4-es portra kerül a 12 bit alsó 8 bitje, azaz 4000 - Int(4000 / 256) = **160**, míg az 5-ös portra a felső 4 bit, azaz Int(4000 / 256) = **15**

Íme a kód, ami megszólaltat egy hangot:

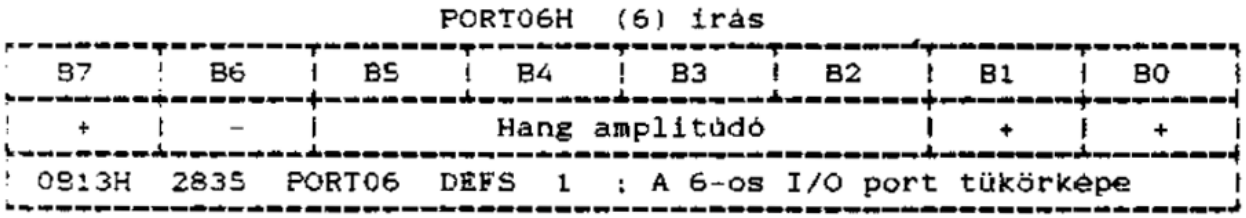
```
LD   HL,2000          ; HL = PITCH értéke, ami most 2000
LD   A,L              ; A = PITCH alsó 8 bitje
OUT  (4),A            ; kiküldjük a 4-es portra
LD   A,(2834)         ; A = 5-ös port tükre
AND  128+64+32+16     ; kinullázzuk az alsó 4 bitet
OR   16               ; Hangjel bit 1-esre állítása
OR   H                ; majd "rákeverjük" a PITCH felső 4 bitjét
LD   (2834),A         ; visszaírjuk a módosult értéket a rendszerváltozóba
OUT  (5),A            ; és kiküldjük az 5-ös portra
```

Persze ez csak akkor fog megszólalni, ha a ***Hangjel*** engedélyezve van az 1. pontban leírtak szerint. Ennek engedélyezését végzi el a pirossal írt sor, amire akkor nincs szükség, ha korábban a *Hangjel* bit már bekapcsolásra került és menet közben nem kapcsoljuk ki, mondjuk két hang közötti szünet miatt.

3. Hangerő (VOLUME)

Még egy paramétert be tudunk állítani és ez a hangerő (VOLUME) avagy az *amplitúdó*. Alapértelmezetten a hangerő a maximális értéken, **15**-ön áll, de sosem tudhatjuk, hogy mi történt a TVC-n mielőtt elindult a programunk, így azt célszerű mindig beállítani magunknak.

A hangerőt a **6-os** porton keresztül tudjuk beállítani, a **2-5 biteken**. Mivel a hangerő **0 – 15** közötti értéket vehet fel, így 4 bit elegendő hozzá.



Kicsit macerás a megfelelő bitmaszk összeállítása amiatt, hogy a 2-5 biteken kell megadnunk a hangerőt anélkül, hogy a többi bitet felülírnánk, de azért semmi extra. A 6-os port rendszerváltozóban levő tükörképét kiolvassuk a decimális **2835** memóriacímről és azt felhasználva készítjük el a megfelelő kiküldendő byte-ot. Íme a kód:

```
LD   A,15          ; A regiszterbe betöltjük a kívánt VOLUME értéket, ez esetben 15-öt
SLA  A             ; mivel a porton a 2-5. bitekre kell írni, ezért 1 bittel eltoljuk balra az értéket
SLA  A             ; majd még 1 bittel, ezt követően a 2-5. biteken lesz a VOLUME értéke
LD   L,A           ; a két bittel balra eltolt hangerő értéket betöltjük az L regiszterbe
LD   A,(2835)      ; kiolvassuk a 6-os port tükrét a rendszerváltozóból
AND  128+64+2+1    ; az alsó és felső két-két bithez nem nyúlunk, a 2-5. biteket töröljük
OR   L             ; A regiszterre OR-al "rákeverjük" a 2 bittel eltolt hangerőt az L regiszterből
LD   (2835),A      ; visszaírjuk a módosult értéket a rendszerváltozóba
OUT  (6),A         ; kiküldjük az eredményt a 6-os portra és készen is vagyunk
```

Ha a hanglejátszásunk egyenletes hangerőn történik, akkor elég egyszer beállítani a hangerőt. Azonban ha a hangeffekteknél a hangerő is számít, akkor azt a műveletet ennek megfelelő gyakorisággal kell elvégezni.

4. Hangkitartás (DURATION)

A fenti hangerő beállító kód után bemásolva a 2. pontban, a hangmagasságnál ismertetett kódot, ahol egy PITCH 2000 frekvenciájú hangot adunk ki, majd ebből a két kódocskából álló forrás lefordítva és lefuttatva a következő BASIC utasítást valósítjuk meg assembly-ben:

SOUND PITCH 2000, VOLUME 15

Vagyis csak majdnem, ugyanis a mi assembly kódunk nemcsak, hogy kiadja a 2000-es frekvenciájú hangot 15-ös hangerőn, de az folyamatosan szólni is fog, amíg ki nem kapcsoljuk a TVC-t, vagy nem nyomunk egy *reset*-et :) Bizony, mivel nekünk kell arról gondoskodnunk, hogy egy kiadott hang meddig szóljon és ha ezt nem tesszük meg, akkor az utoljára kiküldött hang folyamatosan szól a végtelenségig. Ez egyébként alapvetően egy jó dolog, mert ez azt jelenti, hogy amint kiküldtük a portra egy hangot, nincs vele több dolgunk se nekünk, se a TVC-nek mindaddig, amíg egy másik hangot nem szeretnénk

kiadni, vagy kikapcsolni az éppen szóló hangot. Ez így elég CPU takarékos megoldás, szóval kezdjük el megkedvelni :)

De hogy tudjuk a hangot kikapcsolni?

Ez nagyon egyszerű, a *Hangjel*et kell letiltanunk. Ez pont ugyanaz a kód, amivel engedélyeztük, csak most 1-es helyett 0-ra állítjuk a megfelelő bitet. Így:

```
LD   A,(2834) ; A regiszterbe betöltjük az 5-ös port tükrét
AND  128+64   ; a felső 2. bit kivételével a többit töröljük, a Hangjelet is!
LD   (2834),A ;  visszaírjuk az értéket a rendszerváltozóba
OUT  (5),A    ; és kiküldjük az értéket az 5-ös porta
```

Na, akkor ezzel már nem fog örökre szólni a kiadott hang. Másoljuk be az előző két kódrészlet mögé közvetlenül és fordítsuk le. ***Majd próbáljuk is ki!*** Most meg annyi történik, hogy egy nagyon rövid időre megszólalt a hang, aztán semmi. ***Miért?*** Mert annyi ideig szólt a hang, amíg a TVC végrehajtotta az utasításokat és oda nem ért a hang kikapcsoláshoz. És ez gépi kódban elég gyorsan megtörténik.

Szóval valahogy időzítenünk kellene a hang kitartását.

Erre számos módszer van, de nekünk az lenne a szerencsés, hogy ha valami nagyon pontos időzítést tudnánk használni, ami lehetőleg a CPU-t se nagyon terheli le, szóval nem egy egy helyben toporgó ciklus, ami alatt semmi mást nem tudunk csinálni, így a játékunkat / programunkat se tudjuk futtatni.

I. megoldás a ***rendszer timer*** lekérdezése. Ezt a decimális **2845**-ös memóriacímen találjuk és az a jó benne, hogy a **20 ms**-enként végrehajtott megszakítás automatikusan növeli eggyel, tehát óraként, vagy stopperként is lehet használni, ami a TVC bekapcsolásától számolja az eltelt 20 ms-ek számát. Ezzel csak az a baj, hogy mivel ez egy 16 bites cím, így **1311** másodpercenként (65536 * 20 / 1000) lenullázódik és újra kezdi a számolást, ami bekavarhat nekünk, ha nem figyelünk erre oda.

II. megoldás, hogy megvárjuk a képernyőfrissítést, ami egy játék esetében amúgy is célszerű, és szintén **20 ms**-enként következik be. Ha elég gyors a kódunk *(márpedig illik annak lennie)*, azaz egy főciklusa lefut egy képernyőfrissítés alatt, akkor a főciklusunkban valahol elhelyezett **HALT** megadja ezt a **20 ms**-es ütemet a hanglejátszás időzítéséhez.

III. megoldás, hogy a rendszermegszítást végét átirányítjuk a saját megszakítás rutinunkra és az egész hang- és zenelejátszást ebben a megszakításrutinban futtatjuk, amit a TVC 20 ms-enként automatikusan meg fog hívni. A *TVC Boing Ball Demo* ezt a megoldást alkalmazza, ha valakit érdekel, abban meg tudja nézni hogyan.

De próbáljuk ki, hogy a kis programocskánkba tegyük be a hang kikapcsolás elé **5 darab HALT** utasítást, ami ugye azt jelenti, hogy 5*20 = **100 ms**-ig fog szólni a hangunk, ami még mindig csak **0,1** másodperc, de már egy jól hallható időtartam, mielőtt kikapcsolnánk. Ezzel az időzítés működik, mindenünk megvan, hogy hangot játszunk le TVC-n, akár egy zenét is.

Aki lusta volt a kódrészleteket a leírtak szerint összemásolni, annak itt van egyben az egyet prüttyenő kis programunk TVC Studio-ba bemásolható és fordítható változata:

```
                ORG  6639           ; A program kezdőcíme
                ;{BASIC Header
                DB   $0F,$0A,$0,$DD ; "10 PRINT" - basic token
                DB   ' USR'         ; " USR"
                DB   $96            ; "("
                DB   '6659'         ; "6659"
                DB   $95,$FF        ; ")"
                DS   0,0,0,0,0      ; 5 db 0, hogy 6659-nél kezdődjön a kód
                ;}

                ;{egy hang kiadása
SOUND           LD   A,15           ; A = VOLUME értéke
                SLA  A              ; 1 bittel eltoljuk balra az értéket
                SLA  A              ; majd még 1 bittel
                LD   L,A            ; L = a két bittel balra eltolt hangerő érték
                LD   A,(2835)       ; kiolvassuk a 6-os port tükrét
                AND  128+64+2+1     ; a 2-5. biteket töröljük
                OR   L              ; OR-al "rákeverjük" a 2 bittel eltolt hangerőt
                LD   (2835),A       ; visszaírjuk a rendszerváltozóba
                OUT  (6),A          ; kiküldjük az eredményt a 6-os portra
                ; PITCH 2000
                LD   HL,2000        ; HL = PITCH értéke, ami most 2000
                LD   A,L            ; A = PITCH alsó 8 bitje
                OUT  (4),A          ; kiküldjük a 4-es portra
                LD   A,(2834)       ; A = 5-ös port tükre
                AND  128+64+32+16   ; kinullázzuk az alsó 4 bitet
                OR   16             ; Hangjel bit 1-esre állítása
                OR   H              ; majd "rákeverjük" a PITCH felső 4 bitjét
                LD   (2834),A       ; visszaírjuk a rendszerváltozóba
                OUT  (5),A          ; és kiküldjük az 5-ös portra
                ; duration
                HALT                ; várunk
                HALT                ; 5 db
                HALT                ; cursor-megszakítást
                HALT                ; ezzel időzítve
                HALT                ; a hangunkat 100 ms-el
                ; Hangjel ki
                LD   A,(2834)       ; A regiszterbe betöltjük az 5-ös port tükrét
                AND  128+64         ; a felső 2. bit kivételével a többit töröljük
                LD   (2834),A       ; visszaírjuk a rendszerváltozóba
                OUT  (5),A          ; és kiküldjük az értéket az 5-ös porta
                ;}
                RET                 ; visszatérünk BASIC-be
                END
```

Oké, prüttyenni már tudunk, de hogy lesz ebből zene?

Úgy, hogy le kell szépen tárolni a „kottánkat”, majd azon végigmenve lejátszani azt. Ha egyenletes hangerőn játsszuk le a zenénket, akkor az adatokba a **PITCH** és **DURATION** értékeket elegendő felváltva megadni, majd mondjuk egy **0** értékkel jelezni, hogy a zene végére értünk. Ezt követően nincs más dolgunk, mint minden főciklusban – ami egy elhelyezett **HALT** utasítás esetén **20 ms**-enként lefut egyszer - egy hangot megszólaltatni vagy kitartani azt a hangot, amit utoljára megszólaltattunk és mellette vezérelhetjük a játékunkat is. Ha egy hang kitartási ideje lejárt, akkor kiolvassuk a „kottánkból” a következő hangot és megszólaltatjuk. Mindezt addig ismételjük, amíg a kotta végére nem értünk. A **4095**-ös PITCH érték szünet hangnak felel meg.

*A következő forráskód egy az egyben beilleszthető a **TVC Studio**-ba és fordítható, futtatható, csak a **CAS fájl létrehozása** opciót kell bepipálni. **ESC**-re kilép a program BASIC-be.*

```
                ; zenelejátszás példa
                ORG  6639           ; A program kezdőcíme
                ;{BASIC Header
                DB   $0F,$0A,$0,$DD ; "10 PRINT" - basic token
                DB   ' USR'         ; " USR"
                DB   $96            ; "("
                DB   '7000'         ; "7000"
                DB   $95,$FF        ; ")"
                DS   346,$0         ; 346 db 0
                ;}
```

```
                ;{Konstansok
SOUND_PORT_LO   EQU  4                      ; 0-7.bitek: sound PITCH alsó 8 bit
SOUND_PORT_HI   EQU  5                      ; 0-3.bitek: sound PITCH felső 4 bit
VOLUME_PORT     EQU  6                      ; 2-5.bitek: sound VOLUME portja
PORT_5_MEM_MIRROR EQU 2834                  ; a 5-ös port tükörképe a memóriában
PORT_6_MEM_MIRROR EQU 2835                  ; a 6-os port tükörképe a memóriában
KEYBOARD_7      EQU  2904                   ; bill. mátrix 7.sor
KEY_ESC         EQU  8                      ; Esc gomb bitje a mátrixban
                ;}

                JP   INIT

                ;{Változók
MUSIC_POS       DW   0                      ; zene pozíció
DURATION        DB   0                      ; hang kitartásából ennyi van hátra
                ;}

                ;{INIT
INIT            LD   HL,MUSIC
                LD   (MUSIC_POS),HL         ; MUSIC_POS-t a MUSIC elejére állítjuk
                CALL STOP_SOUND             ; hangjel kikapcsolása
                ; Hangerő beállítása
                LD   A,15                   ; A = VOLUME értéke
                SLA  A                      ; 1 bittel eltoljuk balra az értéket
                SLA  A                      ; majd még 1 bittel
                LD   L,A                    ; L = 2 bittel eltolt hangerő érték
                LD   A,(PORT_6_MEM_MIRROR)  ; kiolvassuk a 6-os port tükrét
                AND  128+64+2+1             ; a 2-5. biteket töröljük
                OR   L                      ; OR-al "hozzáadjuk" a hangerőt
                LD   (PORT_6_MEM_MIRROR),A  ; visszaírjuk a rendszerváltozóba
                OUT  (VOLUME_PORT),A        ; kiküldjük az eredményt a 6-os portra
                ;}

                ;{MAIN_LOOP - főciklus
MAIN_LOOP       HALT                        ; várakozás egy megszakításra
                ; Zene lejátszása
                LD   A,(DURATION)           ; A = hang kitartásból hátralevő idő
                OR   A                      ; A = 0?
                JP   NZ,SOUND_DURATION      ; még nem nulla, ugrás hangkitartásra
                ; következő hang a zenéből
                LD   HL,(MUSIC_POS)         ; HL = PITCH a kottából
                LD   A,(HL)                 ; A = PITCH alsó byte a kottából
                LD   E,A                    ; elrakjuk E regiszterbe
                INC  HL
                LD   A,(HL)                 ; A = PITCH felső byte a kottából
                LD   D,A                    ; DE = PITCH
                INC  HL
                LD   A,(HL)                 ; A = DURATION a kottából
                LD   (DURATION),A           ; elmentjük a változóba
                INC  HL                     ; átlépjük a DURATION adatokat
                INC  HL                     ; HL a következő hangra mutat
                LD   (MUSIC_POS),HL         ; Music pozíció mentése
                ; Zene végére értünk vizsgálat
                OR   A                      ; DURATION = 0?
                JP   NZ,SOUND_OUT           ; ha nem, akkor kiadjuk a hangot
                CALL STOP_SOUND             ; ha igen, akkor vége a kottának
                ; ugrás vissza a Zene  elejére
                LD   HL,MUSIC
                LD   (MUSIC_POS),HL         ; MUSIC_POS = a zene eleje
                JP   OTHER_CODE             ; ugrás a program további részére
                ; hang kiadása
SOUND_OUT       LD   A,E                    ; A = PITCH alsó 8 bitje
                OUT  (SOUND_PORT_LO),A      ; kiküldjük a 4-es portra
                LD   A,(PORT_5_MEM_MIRROR)  ; A = 5-ös port tükre
                AND  128+64+32              ; kinullázzuk az alsó 4 bitet
                OR   16                     ; Hangjel bit 1-esre állítása
                OR   D                      ; majd hozzáadjuk a PITCH felső 4 bitjét
                LD   (PORT_5_MEM_MIRROR),A  ; visszaírjuk a rendszerváltozóba
                OUT  (SOUND_PORT_HI),A      ; és kiküldjük az 5-ös portra
                JP   OTHER_CODE             ; ugrás a program további részére
                ; hangkitartás időzítése
SOUND_DURATION  DEC  A                      ; DURATION = DURATION - 1
                LD   (DURATION),A           ; elmentjük a változóba
                ; ide írhatjuk a játékunk kódját
OTHER_CODE      NOP
                ; ESC gombra kilépés
                LD   A,(KEYBOARD_7)         ; A = keyboard mátrix 7
                CP   KEY_ESC                ; ESC-et nyomtak?
                JP   Z,THE_END              ; ha igen, akkor ugrás a kilépésre
                JP   MAIN_LOOP              ; vissza a főciklus elejére
                ;}

                ;{THE_END
THE_END         CALL STOP_SOUND             ; hangjel kikapcsolása
                RET                         ; visszatérünk BASIC-be
                ;}
```

```
                ;{STOP_SOUND - HANGJEL tiltása
STOP_SOUND      LD   A,(PORT_5_MEM_MIRROR) ; A = 5-ös port tükre
                AND  128+64                ; töröljük a "HANGJEL" bitet
                LD   (PORT_5_MEM_MIRROR),A ; visszaírjuk a rendszerváltozóba
                OUT  (SOUND_PORT_HI),A     ; majd kiküljük a portra
                RET                        ; vissza a hívóhoz
                ;}

                ;{MUSIC - a zenénk hangjai: PITCH, DURATION párok egymás után
MUSIC           DW   3349,10,3503,10,3349,10,3503,10,3598,20,4095,1,3598,20
                DW   3349,10,3503,10,3349,10,3503,10,3598,20,4095,1,3598,20
                DW   3723,10,3701,10,3652,10,3598,10,3537,20,3652,20
                DW   3598,10,3537,10,3503,10,3431,10,3349,20,4095,1,3349,20
                DW   4095,40   ; 4095 PITCH érték szünetet jelez
                DW   0,0       ; 0 DURATION érték jelzi a zene végét
                ;}

                END
```