
Programy
dla Atari
V

Vydává 487. ZO Svazarmu — ATARI KLUB v Praze 4.
Šéfredaktor a vedoucí redakční rady JUDr. Jan Hlaváček.
Zástupce šéfredaktora ing. Stanislav Borský.
Obálku navrhl RNDr. J. Tamchyna.
Adresa redakce:
487. ZO Svazarmu - ATARI KLUB Praha
REDAKCE
poštovní přihrádka 51
100 00 P r a h a 1 0
Řídí redakční rada: V. Bílek, ing. J. Biskup, RNDr. J. Bok, CSc., ing. S. Borský, ing. V. Friedrich, ing. O. Hanuš, RNDr. L. Hejna, CSc., Z. Lazar, prom. fyz., CSc., ing. M. Vavrda.

Vychází šestkrát ročně. Neprodejné. Členům klubu distribuováno zdarma. Nepravidelné přílohy na objednávku jsou kompenzovány zvláštním klubovým příspěvkem.
Rozsah čísla 116 stran. Neprošlo jazykovou úpravou.
TISK ČTK-REPRO
Do tisku předáno v 7/88
Vydávání schváleno OV Svazarmu Praha 4 a OŠK ONV Praha 4.
Evidenční číslo ÚVTEI 86 042.

MARK CHASIN

Programování v asembleru pro počítače ATARI

Překlad ing. Karel Šmuk

Technická příprava ing. Lubomír Bezděk

Revize překladu ing. Petr Jandík

Kapitola 1:

Úvod

BASIC je asi 200x pomalejší než strojové programy. Všechny počítače ATARI od 400 ke 1450XLD jsou kompatibilní z hlediska assembleru.

Assembler je prostředek pro zápis programů ve strojovém kódu. Hlavní výhodou takto zapsaného programu je jeho rychlost. Ve skutečnosti je možno napsat program v assembleru, který je více než 1000x rychlejší než jeho ekvivalent v BASICu.

Další výhodou je to, že v assembleru má programátor plně v ruce všechny možnosti počítače. V BASICu je programátor často odstíněn od základních funkcí, které je možno využívat pouze prostřednictvím jazyka assembler, nebo strojového kódu.

Nevýhody:

- -je nutné se naučit další programovací jazyk
- -při každé změně je nutno znovu přeložit program do strojového kódu
- -mnoho instrukcí i pro nejjednodušší úkoly - z toho plyne, že programy v assembleru jsou obvykle velmi dlouhé
- -obtížnost porozumět výpisu programu (důvod pro to, abychom programy bohatě vybavovali poznámkami)
- -nutná podrobná znalost architektury počítače, organisace paměti a OS a vazeb na hardware.

Práce s jazykem assembler

Pro převod programu v assembleru do strojového jazyka používáme program nazývaný ASSEMBLER(překladač jazyka assembler).Pro počítače ATARI existuje několik assemblerů,popsané techniky budou použitelné s kterýmkoliv z nich.Výpisy programů v této knize byly vytvořeny pomocí Cartridge ASSEMBLER/EDITOR verse ATARI. V kapitole 6 jsou popsány změny,nutné pro použití programů s jinými assemblery firmy.

Překladače

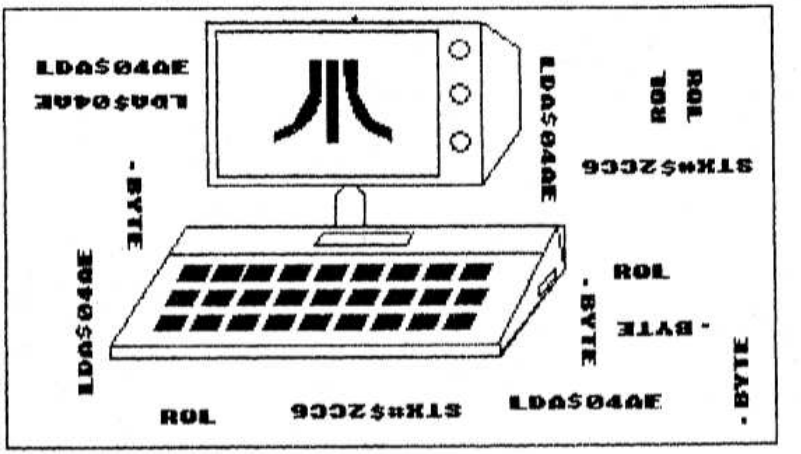
Existují i možnosti přeložit program ve vyšším jazyce(např.BASIC) do strojového kódu, který bude 5 až 10 krát rychlejší než při interpretaci,ale obecně program napsaný v assembleru bude mnohem rychlejší. Další nevýhoda kompilovaného programu je jeho velikost.Např.některé podprogramy v kapitole 7 jsou dlouhé okolo 100 bytů,ale tytéž programy napsané v BASICu a zkompilované mohou zabrat až 8000 bytů a sotva je lze použít jako podprogramy pro jiný program v BASICu.

Terminologie

RAM-paměť s přístupem pro čtení i zápis
ROM-paměť pevná, lze pouze číst
OS-operační systém, zapsaný v ROM, řídící téměř vše co se děje v počítači. Bez operačního systému by se při zapnutí počítače nestalo vůbec nic. Dále se naučíme jak s OS komunikovat. Operační systémy jednotlivých počítačů se vzájemně liší, ale určité funkce jsou ve všech dostupné pomocí tzv. vektorů, které jsou shodné pro všechny OS na 8 bitových počítačích ATARI. Vektory představují odkazy, podle nichž se najdou určité subrutiny v operačním systému. Části OS je možné využívat i bez použití vektorů, pak ale není záruka, že takový program poběží i na jiném počítači ATARI. Proto se důrazně varuje před takovými praktikami.

DOS-diskový operační systém. Program, který řídí diskové jednotky a jejich připojení k počítači. DOS 2.5, který je jedním z nejčastěji používaných DOS u ATARI, je složen ze dvou částí DOS.SYS a DUP.SYS. DOS.SYS se natáhne do počítače při prvním zapojení a je trvale přítomen. DUP.SYS se natáhne pouze při specifikaci DOS z klávesnice. DOS umožňuje řadu funkcí, viz menu při ohlášení programu.

Kapitola 2:



Začínáme

Hexadecimální systém

Číslice 0,1,2,. 9,A,B,C,D,E,F
při zápisu hex. čísla použijeme znak pro hex. číslo $, např. $B7

Organizace dat

1 byte 256 možností
2 byte 65536 "
3 byte 16777213 "

Znaménko čísla

Při dvoubytové aritmetice celých čísel obsahuje znaménko pouze nejvyšší bit, tj. máme 1 znaménkový bit a 15 významových.

Adresování paměti

Adresový prostor ATARI je 65536 bytů, tj. tolik, kolik může být adresováno 2 byty. Více paměti může být adresováno nepřímým způsobem.

Kapitola 3:

Hardware ATARI

Počítače ATARI a řada dalších používají jako procesor/CPU/ obvod 6502. Mluvíme-li o strojovém jazyce, myslíme tím přímé, či nepřímé programování procesoru 6502.

V ATARI jsou dále 3 specielní obvody, též mikroprocesory, které nejsou v žádném jiném počítači a sice ANTIC, POKEY a GTIA, které ve spolupráci s 6502 vytvářejí grafiku a zvukové efekty. Použití těchto obvodů bude popsáno dále. Počítač ATARI 130 XE obsauje navíc obvod pro řízení paměti FREDY. Nejdříve se seznámíme s jednotlivými částmi 6502:

AKUMULÁTOR

První částí CPU je akumulátor, v assembleru obvykle označen A. V něm se provádějí vlastní výpočty - porovnávání, sčítání, odčítání atd. Při provádění jakékoliv operace musíme obvykle načíst hodnotu, která se má změnit, do akumulátoru,změnit ji a případně uložit do paměti.

Registry X a Y

Registry X a Y jsou částí CPU, nejsou přímo dostupné z BASICu, pouze ze strojového jazyka. Můžeme je použít dvěma způsoby. Buď pro krátkodobé uchování informace, nebo pro snazší přístup k informaci, uložené v po sobě jdoucích buňkách paměti.

Programový čítač PC

Programový čítač je dvakrát delší než ostatní registry v 6502, tj. 2 byty namísto jednoho. PC obsahuje adresu, která se bude provádět jako následující.

Ukazatel zásobníku

Zásobník si můžeme představit jako balíček karet, do kterého smíme přidat, nebo si vzít pouze kartu, která leží nahoře.

V 6502 se jako zásobník používá první stránka paměti, tj. adresy 256 - 511 decimálně, nebo $100 - $1FF hex. včetně. Ukazatel zásobníku je částí 6502 a obsahuje informaci o tom, co je zrovna na vrcholu zásobníku / přesněji na dně, protože zásobník se zvětšuje směrem ke klesajícím adresám/. Nemusíme se starat o to, kolik hodnot je v zásobníku nebo jak přidat další, o to se stará 6502. Musíme však dbát na to, abychom do zásobníku neuložili více než 256 hodnot.

Registr stavu procesoru

Jako stavový registr označujeme 1 bytový soubor různých bitů,

které 6502 používá pro registraci některých příznaků. Tyto příznaku /flags/ jsou obvykle označovány jednopísmenovými zkratkami:

písmeno	označení	význam
C	Carry	Přenos
Z	Zero	1=výsledek je nulový
I	IRQ Disable	1=disable, zákaz přerušení
D	Decimal mode	1=dekadický modus
B	Break command	přerušení instrukcí BRK
V	Overflow	1=přetečení
N	Negative	1=výsledek je záporný

Stavový registr: N V — B D I Z C

Příznak přetečení: tento bit udává, zda v předchozí operaci došlo k přenosu. Např. výsledek sčítání dvou bytů je větší než 255. Používá se téměř ve všech aritmetických instrukcích.

Příznak nuly: bit Z nám říká, zda výsledek operace byl nulový, pokud ano Z=1.

Příznak IRQ: toto znamená Interrupt request, nebo-li požadavek přerušení. Pokud-je tento bit roven 0, počítač reaguje na přerušení. Pokud je I=1 je přerušení maskováno/zakázáno/.

Příznak dekadického modu : 6502 má dva mody, v nichž provádí aritmetické operace, binární a dekadický. D=1 znamená, že operace jsou v dekadickém modu, D=0 znamená binární modus. Obecně většina operací v assembleru pracuje s binárními hodnotami.

Příznak přerušení instrukcí BRK : tento bit je nastavován pouze 6502 a programátor jej nemůže změnit. Používá se k určení, zda přerušení bylo způsobeno instrukcí BRK. Protože nemůže být ovládán programátorem, v běžných programech nemá velké použití. Používá se v ladicích programech.

Příznak přetečení: i když každý byte má 8 bitů, je při aritmetických operacích v binární matematice se znaménkem nejvyšší bit použit jako znaménko. Bit přetečení signalisuje, že při aritmetické operaci došlo k přesunu do znaménkového bitu. V=1 znamená přetečení. Tento bit můžeme testovat, abychom měli jistotu, že výsledek bude interpretován jako správné binární číslo se znaménkem.

Příznak záporné hodnoty : posledním ve stavovém registru je bit, signalizující zápornou hodnotu. Pokud je N=1, byl výsledek předchozí operace záporný, pokud je N=0, byl výsledek kladný nebo 0. Rozlišení mezi nulovou a nenulovou kladnou hodnotou provedeme pomocí bitu Z.

Testy pomocí bitů N,C,Z představují hlavní prostředek pro řešení programů v assembleru.

Systém rozdělení paměti

Paměť v počítači s procesorem 6502 je rozdělena na stránky po 256 bytech. Pravděpodobně jste tento termín již slyšeli ve spojitosti se stránkou 6, rezervovanou pro použití programátora v BASICu. Stránka 6 je oblast paměti od $600 do $6FF, nebo dekadicky od 1536 do 1791. Atari zaručuje, že systémový software tuto část paměti nevyužívá. Za určitých podmínek však může dojít k přetečení stránky 5 a přesunu informací do stránky 6. Proto používejte stránku 6 opatrně s vědomím možných problémů.

I některé další stránky mají specifické použití. Nejdůležitější je stránka 0, prvních 256 byte v počítači. Stránka 0 má zvláštní význam při programování v assembleru, protože přístup do této stránky je rychlejší než kamkoliv jinam v paměti a některé operace mohou být prováděny pouze s použitím buněk stránky 0. Většina stránky 0 je však použita pro operační systém a pokud používáte Basic nebo assembler Editor, je k disposici pouhých 6 bytů. 6 bytů je málo a proto se naučíme několik triků, jak použít více bytů ze stránky 0, nebo jak hospodárně využít těch, které máme k disposici.

Stránky 2-5/$200 - $5FF dek. 512 - 1535/ obsahují informace potřebné pro operační systém. Oblast nad stránkou 6 je obecně rezervována pro DOS. Paměť, kterou může programátor použít bez rizika přepsání programu DUP. SYSem obecně začíná na $3200 resp. dek. 12800.

Kapitola:4

Názvosloví

Zápis čísel - kdykoliv použijeme v instrukci assembleru číslo, musíme mu předřadit značku čísla #. Musíme pečlivě rozlišovat mezi číslem a obsahem adresy v počítači. Jejich vzájemná záměna může způsobit ty nejhorší chyby v programu, kdy můžeme hledět na program řadu dní, aniž bychom chybu zpozorovali. Pokud nebude před číslem nic, považujeme je za dekadické. Chceme-li použít hexadecimální zápis, použijeme návěstí $. Např. adresa paměti $11 odpovídá v dekadické verzi 17. Číselnou konstantu označíme #. Většina assemblerů rozlišuje ještě binární zápis ä. Např. #ä110101.

Instrukční soubor 6502 - každá instrukce je podrobně probrána v příloze 1. Zde se o nich stručně zmíníme, abychom se seznámili s jejich názvy i použitím. Každá instrukce je třípísmenovou zkratkou anglického názvu instrukce. Této zkratce se říká mnemonika. Způsobům, jak tyto instrukce adresují paměť, bude věnována kapitola 5. V této části pohovoříme o skupinách instrukcí s důrazem na to, jak mohou být použity při programování.

Instrukce typu LOAD

v této skupině jsou tři instrukce:

```
LDA  naplň Akumulátor
LDX  -"-   registr X
LDY  -"-   registr Y
```

Tyto instrukce se používají k naplnění příslušného registru z buňky paměti, např. LDA $0243, naplní akumulátor obsahem buňky na adrese $0243. Instrukce LOAD nemění obsah buňky v paměti, příslušná buňka obsahuje stejnou hodnotu před i po provedení instrukce LDA. Protože víme, že veškeré aritmetické operace se provádějí v akumulátoru, je jedno použití instrukce LDA zřejmé.

Ukládací instrukce<store>

```
STA  ulož Akumulátor
STX  -"-  registr X
STY  -"-  registr Y
```

typická instrukce vypadá takto:

```
STA $0243
```

Tato instrukce píše obsah akumulátoru do buňky s adresou $0243. Obsah akumulátoru,nebo příslušného registru se přitom nemění. Chceme-li např. uložit číslo 8 do čtyř různých buněk paměti, můžeme použít následující program:

```
LDX #8 ;dej do registru X hodnotu 8
STX $CC ;ulož do první buňky
STX $CD ;      -"-
```

```
STX $12 ;       -"-
STX $D5 ;     hotovo
```

hodnotu 8 v registru X jsme nemuseli znovu naplňovat, zůstane tam až do chvíle, kdy obsah registru X změníme. Pro stejný úkol můžeme použít i akumulátor nebo registr Y.

Velmi časté použití instrukcí LOAD a STORE je přesun hodnot z jedné nebo více buněk paměti jinam. Např.

```
LDA $5983 ;vezmi první hodnotu
STA $0243 ;ulož jinam
LDA $0876 ;vezmi další
STA $0341 ;ulož
atd.
```

Řídicí instrukce

Posloupnost vykonávání instrukcí programu vám umožňují změnit dva typy instrukcí.Jsou to instrukce JUMP <skok> a BRANCH <větvení>.

Instrukce JUMP.

```
JMP skok na specifickou adresu.
JSR skok do podprogramu.
```

Přenos řízení je nepodmíněný, t.j. bude proveden za jakýchkoliv okolností. Při předání řízení instrukcí JMP, bude výpočet prováděn od adresy, která je v ní uvedena.Při předání řízení instrukcí JSR bude výpočet rovněž pokračovat na uvedené adrese, ale při dosažení instrukce RTS se řízení vrátí bezprostředně za instrukci JSR.

Obdobná k instrukci RTS <návrat z podprogramu> je instrukce RTI <návrat z přerušení>, o ní více později.

Instrukce BRANCH

Na rozdíl od předchozích přenosů řízení má 6502 sadu instrukcí pro podmíněné předání řízení v závislosti na výsledcích předchozích instrukcí.Jsou to:

```
BCC  odskok při C= 0
BCS     _ " _   C= 1
BEQ     _ " _   Z= 1
BMI     _ " _   N= 1
BNE     _ " _   Z= 0
BPL     _ " _   N= 0
BVC     _ " _   V= 0
BVS     _ " _   V= 1
```

Každá z těchto instrukcí závisí na hodnotě některého bitu ve stavovém registru.

```
Příklad: LDA #0  ;inicializuj akumulátor
         BCC SUB4;odskok, je-li C=0
         LDA #1  ;pokud C=1 vezmi #1   do akumulátoru
    SUB4 STA $0243;ulož hodnotu zde
```

V závislosti na bitu C <přenos>je do buňky $0243 uložena 0 nebo

1.Jestliže je C=0, provedl se odskok na SUB 4 a do buňky $0243 byla vložena 0. Pokud je C=1, odskok se neprovede a do $0243 se uloží 1.

Instrukce pro stavový registr

Pomocí těchto instrukcí se manipuluje s jednotlivými bity ve stavovém registru.

```
CLC dosaď C=0
CLD  "   D=0
CLI  "   I=0
CLV  "   V=0
SEC  "   C=1
SED  "   D=1
SEI  "   I=1
```

Aritmetické a logické instrukce

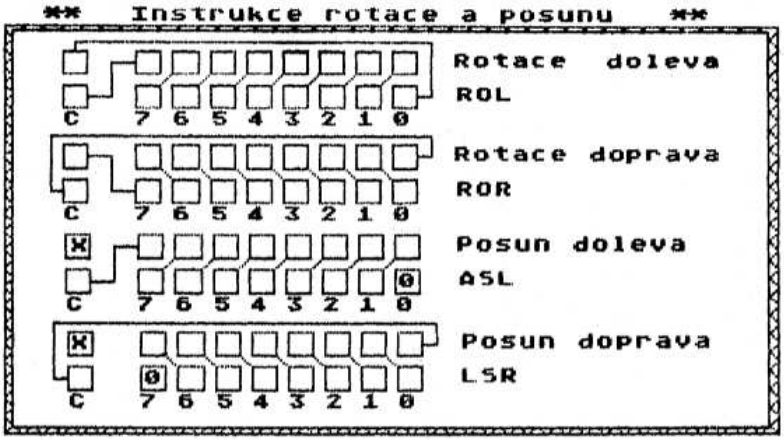
V této skupině jsou všechny instrukce, které provádějí nějaké výpočty.

```
ADC sčítání s přenosem
AND logický součin
ASL aritmetický posuv doleva
BIT testuj bity v paměti a akumulátoru
EOR exkluzivní OR
LSR logický posuv doprava
ORA logický součet
ROL rotace doleva
ROR rotace doprava
SBC odečítání s přenosem
```

Detailní popis těchto instrukcí je v příloze 1.
Instrukce ADC je základní sčítací instrukce 6502. Instrukce sečte hodnotu v akumulátoru, přičte bit C a hodnotu adresovanou instrukcí ADC. Např: Sečteme obsah buňky $0434 s obsahem $0435 a výsledek uložíme do $0436.

```
CLC       ;vynuluj bit C
LDA $0434 ;vezmi 1.číslo
ADC $0435 ;přičti2.číslo
STA $0436 ;ulož výsledek
```

V této skupině jsou 4 instrukce posuvu: ASL,LSR,ROL,ROR. Všechny tyto instrukce posouvají bity, ale různým způsobem. Obě instrukce ROR a ROL použijí bit C stavového registru a posunou kruhově 9 bitů(8 bitů adresového čísla a 1 bit C) doleva nebo doprava.



Instrukce ASL a LSR pracují obdobně s tím rozdílem že, bez ohledu na původní obsah C se do uvolněné pozice v čísle vždy dosadí 0.

Těchto instrukcí se často využívá k násobení nebo dělení dvěma, protože posuv o 1 číslo doleva nebo doprava v znamená v binární aritmetice násobení nebo dělení dvěma.

Tři logické instrukce AND,EOR a ORA pracují s jednotlivými bity. Oba argumenty se porovnávají bit po bitu a výsledný bit obsahuje 0 nebo 1 v závislosti na původních argumentech.

Instrukce AND říká: Pokud oba bity byly 1, je výsledek 1, jinak 0.

Instrukce EOR znamená: Výsledek je 1 , pokud bity operandů na příslušné posici jsou různé. Jsou-li stejné, je výsledek 0.

Instrukce ORA znamená: Ve výsledku je 0, pokud oba bity operandů jsou 0, v opačném případě je výsledek 1.

Poslední instrukcí této skupiny je BIT, což je testovací instrukce. BIT dosadí do bit příznaku N stavového registru hodnotu bitu 7 testovaného čísla.Bit V nabude hodnotu bitu 6 testovaného čísla. Bit Z bude dosazen v závislosti na výsledku logického součinu <AND> testovaného čísla a obsahu akumulátoru. Číslo v akumulátoru se přitom nezmění. Použitím některé z instrukcí BRANCH pak můžeme v programu odpovídajícím způsobem pokračovat.

Manipulační instrukce

Podobně, jako instrukce LOAD a STORE, se následující instrukce používá k přesunům dat z jedné části procesoru nebo paměti do jiné.

PHA	ulož obsah akumulátoru do zásobníku
PHP	ulož stavový registr do zásobníku
PLA	vyber hodnotu ze zásobníku do akumulátoru
PLP	vyber hodnotu ze zásobníku do stavového registru
TAX	přesuň hodnotu akumulátoru do registru X
TAY	přesuň hodnotu akumulátoru do registru Y
TSX	přesuň obsah ukazatele zásobníku do registru X
TXA	přesuň obsah registru X do akumulátoru
TXS	přesuň obsah registru X do ukazatele zásobníku
TYA	přesuň obsah registru Y do akumulátoru

Funkce těchto instrukcí je zřejmá, slouží k přenosu informací mezi jednotlivými registry nebo zásobníkem. Při manipulaci se zásobníkem instrukcemi PHA, PHP, PLA, PLP se automaticky mění také hodnota ukazatele zásobníku.

Instrukce Increment a Decrement

Instrukce v této skupině slouží ke zvýšení nebo snížení hodnoty v paměti nebo registru o 1.

```
DEC  zmenši hodnotu v buňce paměti o 1.
DEX  zmenši hodnotu v registru X o 1.
DEY  zmenši hodnotu v registru Y o 1.
INC  zvětši hodnotu v buňce paměti o 1.
INX  zvětši hodnotu v registru X o 1.
INY  zvětši hodnotu v registru Y o 1.
```

Příklad:

```
LDA #3     ;začni s 3
STA $0243  ;ulož na $0243
INC $0243  ;teď jsou tam 4
INC $0243  ;teď je tam 5
DEC $0243  ;jsou tam zase 4
```

Všimněme si, že neexistuje zvětšování nebo snižování hodnoty akumulátoru; ke zmenšení nebo zvětšení hodnoty musíme použít ADC nebo SBC.

Porovnávací instrukce

Porovnání dvou hodnot nám umožňují 3 instrukce.

```
CMP  porovnej akumulátor s číslem.
CPX   _ " _  registr X s číslem.
CPY   _ " _  registr Y s číslem.
```

Způsob, jak tyto instrukce ovlivní stavový registr, je přesně popsán v příloze 1, uvedeme pouze jednoduchý příklad:

```
LDA $0243  ;vezmi 1.číslo
CMP $0244  ;porovnej je s druhým
BNE SVB6   ;jdi na SVB6 jsou-li čísla různá
LDA #1     ;jsou-li stejná, pokračuj zde
```

Zbývající instrukce

Zbývající instrukce nepatří do žádné skupiny, jsou to BRK a NOP. BRK se využívá hlavně při ladění programů. Způsobí přerušení programu. Spouštíme-li program pomocí některého z ladících programů <DEBUG>, obvykle se nám vypíší obsahy registrů a další relevantní informace. Instrukce NOP nedělá nic, její hlavní význam je rezervovat místo v programu pro případnou další změnu. Při programování v assembleru je umístění instrukce často kritické a instrukce NOP v programu mohou být nahrazeny jiným příkazem aniž by bylo nutno instrukce posouvat.

Tímto končí stručný úvod do instrukčního souboru 6502. Detaily o instrukcích je možno najít v příloze 1 a důrazně doporučujeme pečlivé prostudování celé přílohy.

Poznámky

Kapitola 5:

Způsoby adresování

Nejprve si řekněme, co rozumíme adresovámím. Je to způsob jak procesoru udat objekt, s nímž má pracovat.

Adresování může být přímé nebo nepřímé.
Adresa může být absolutní nebo relativní.
Způsob adresování je implikován <určen> instrukcí.

Adresové mody 6502

6502 může adresovat operand 13-ti různými způsoby. Pro vysvětlení prvních použijeme instrukci LDA, i když tyto adresní mody může využívat i řada dalších instrukcí.

Bezprostřední adresování

Při bezprostředním adresování je operand přímo částí instrukce, t.j. např.:LDA #$4F říká počítači, že má do akumulátoru dodat hodnotu $4F. Totéž bychom docílili pomocí LDA #79.

Každá instrukce v modu bezprostředního adresování zabere v paměti 2 byty: jeden pro kód instrukce a druhý pro vlastní operand. Z toho plyne, že operandem může být pouze číslo mezi 0 a 255.

Dobu vykonávání instrukce udáváme počtem cyklů, které jsou k provedení instrukce zapotřebí. Jeden cyklus je nejkratší časový interval s nímž počítač zachází. Doba jedoho cyklu počítačů ATARI je 560 ns.

Instrukce LDA v bezprostředním modu potřebuje pro vykonání 2 cykly, což je cca. 1 mikrosekunda.

Absolutní adresování

Druhý adresovací modus je absolutní adresování, které použijeme, chceme-li naplnit akumulátor z určité známé buňky paměti.Forma instrukce je:

LDA 315 nebo LDA $13B

čímž říkáme počítači, aby do akumulátoru přenesl obsah buňky na na adrese 315.

Jestliže v buňce 315 byla např. hodnota 243, bude po provedení této instrukce akumulátor obsahovat rovněž 243.

Absolutní adresování vyžaduje 3 byty kódu, protože pro vlastní adresu potřebuje 2 byty. Pořadí bytů adresy u 6502 je: dolní byte <adresa uvnitř stránky>, horní byte <číslo stránky>; toto pořadí je u některých jiných procesorů obrácené.

Po překladu instrukce do strojového kódu se jako první objeví kód instrukce < pro LDA v modu abs. adresování je to $AD> a následuje adresa v pořadí dolní-horní byte:

LDA $2F3C se přeloží AD3C2F

Při prohlížení výpisu z assembleru, nebo prohlížení obsahu paměti to zprvu poněkud mate, ale po kratší praxi si jistě zvyknete.

Adresování v nulté stránce

Tento modus se používá pro naplnění akumulátoru hodnotou některé z prvních 256 buňek paměti, ze stránky 0. Protože žádná z adres v této stránce není vyšší než 255, pro adresování stačí 1 byte a celá instrukce zabere pouze 2 byty.Např.:

LDA $2D nebo LDA 45

Je zajímavé si všimnout, že na provedení instrukce v tomto modu je zapotřebí pouze 3 cyklů proti 4 cyklům při absolutním adresování. Proto v programech, vyžadujících max. rychlost, užívají programátoři stránku 0, pokud je to jen možné. Pozor ovšem na to, že ve většině situací je ve stránce 0 k dispozici jen málo buněk, které můžeme použít.

Indexované adresování v nulté stránce

Toto je první adresní modus, který používá X-registr jako indexový registr. Efektivní adresu získáme jako součet adresy, specifikované v instrukci a obsahu registru X. Pro ilustraci předpokládejme, že v registru X je hodnota 5 a má být provedena instrukce:

LDA $43,X

První část instrukce by sama o sobě znamenala naplnění akumulátoru obsahem buňky $43 ve stránce 0. Pro získání správné adresy k této základní adrese musíme přičíst obsah registru X a dostáváme výslednou adresu $48. Tudíž v tomto okamžiku instrukce znamená naplnění akumulátoru z buňky $48.

Všiměme si toho, že říkáme " v tomto okamžiku". Pokud by obsah registru X byl jiný, např. 2, znamenala by táž instrukce naplnění akumulátoru z buňky o adrese <$43+2>= $45.

Mělo by být zřejmé, že použijeme-li hodnotu z určité buňky ve stránce 0 a pak chceme použít hodnotu z buňky následující, můžeme buď zvýšit hodnotu v registru X nebo základní adresu. To znamená, že můžeme ponechat původní hodnotu v registru X na 2 a napsat:

```
    LDA $44,X
```

a nebo zvýšit hodnotu v registru X na 3 a použít:

```
    LDA $43,X
```

V obou případech bude akumulátor naplněn z buňky $46. Indexované adresování v nulté stránce vyžaduje 2 byty v paměti a trvá celkem 4 cykly .

Absolutní indexované adresování

Další 2 adresovací módy, jsou si tak podobné, že je probereme současně. Jsou příbuzné módu, který jsme si právě popsali a liší se pouze tím, že jsou použitelné pro libovolnou adresu v paměti a ne pouze na stránce 0. Jsou to adresní módy absolutní,X a absolutní,Y.

Obsah registru X resp. Y se přičte k základní adrese, čímž se získá efektivní adresa. Např. předpokládejme, že registr X obsahuje hodnotu 3, pak instrukce:

LDA $0342,X

naplní akumulátor z buňky na adrese <$0342+3>= $0345. To platí i pro registr Y.

Protože pro adresování celé paměti potřebujeme 2 byty, jsou obě tyto instrukce 3 bytové a každá vyžaduje 4 cykly, což je zajímavé, protože absolutní instrukce LDA vyžaduje rovněž 4 cykly. Takže zde za rozšíření možností nic neplatíme. Nebo téměř nic. V případě, že základní adresa plus hodnota v indexním registru ukazují na stránku o 1 vyšší než základní adresa, je zapotřebí 1 cyklus navíc.

Implikované adresování

Všechny instrukce používající implikovaného adresování, jsou jednobytové a kódem instrukce je zároveň určen <implikován> operand <nebo operandy>.

K těmto instrukcím patří instrukce k ovládání příznaků stavového registru, přesuny mezi registry vzájemně, nebo mezi akumulátorem a zásobníkem, instrukce návratu RTS a RTI a dále instrukce BRK a NOP. Většině stačí k vykonání 2 cykly, ale některé trvají déle <viz příloha 1>.

Relativní adresování

Relativní adresování je používáno všemi instrukcemi větvení. Druhý byte instrukce je interpretován jako celé číslo se znaménkem v rozmezí -128 až +127. Tato hodnota se přičte k adrese instrukce následující za instrukcí větvení. Je-li splněna podmínka, vyžadovaná instrukcí, provede se skok na vypočtenou adresu. Není-li podmínka splněna, pokračuje se instrukcí následující za instrukcí větvení.

Relativní adresování umožňuje větvení pouze v poměrně úzkém okolí testovací instrukce, zato není závislé na absolutní adrese uložení programu a má proto značný význam v relokatibilních programech <schopných funkce nezávisle na adrese uložení v paměti>.

Nepřímé absolutní adresování

Tento způsob adresování využívá jediná instrukce - nepřímý skok. Více o ní v příloze 1.

Dva nepřímé mody

Poslední dva způsoby adresování instrukcí jsou nepřímé. Oba tyto způsoby se často zaměňují, protože jsou si velice podobné, ale jejich použití je zcela rozdílné. První z nich je NEPRIME INDEXOVANE ADRESOVANI, např:

```
LDA ($43),Y
```

Závorky v zápisu označují, že jde o nepřímou adresu. Instrukci můžeme iterpretovat následovně.

1. Ve stránce 0, v buňkách $43, $44 najdi 2 bytovou hodnotu.
2. tuto hodnotu považuj za adresu v paměti.
3. K této adrese přičti hodnotu z registru Y.
4. Naplň akumulátor z buňky o takto získané adrese.

Udělejme si podrobnější příklad:
Předpokládejme, že v buňce $43 je hodnota $53 a v buňce $44 hodnota $E4. Dále předpokládejme, že registr Y obsahuje hodnotu 6:

Místo	obsah
$43	$53
$44	$E4
Y-registr	6

Nyní máme provést instrukci LDA ($43),Y. Procesor nejprve zjistí hodnotu na buňkách $43, $44 a považuje ji za adresu, uloženou v pořadí dolní/horní byte, t.zn., že vlastní adresa je $E453. Pak k této adrese přičte hodnotu z registru Y (t.j. 6) a vypočte efektivní adresu $E459.
I když to vypadá nepřehledně, řada aplikací by bez tohoto modu byla jen obtížně naprogramovatelná.
Poněvadž tento modus používá adresu ze stránky 0, zabírá v paměti pouze 2 byty. Výpočet je však poměrně komplikovaný a proto provádění instrukce potřebuje 5 cyklů.

Poslední adresovací modus je INDEXOVANE NEPRIME ADRESOVANI, které se často plete s předchozím módem. Jeho použití je však zcela jiné. Typická instrukce je např.:

```
LDA ($43,X)
```

Všimněme si, že instrukce používá registr X namísto Y a že závorky uzavírají celý operand včetně X. Tato instrukce bude procesorem 6502 intepretována následovně:

1. Přičti hodnotu v registru X k základní adrese $43 (t.j. pokud X= 4, výsledkem je $47)

2. Tento součet je považován za další adresu ve stránce (0 v tomto případě $47)
3. Najdi 2bytovou hodnotu na této a následující adrese ($47,$48) a interpretuj ji jako novou adresu.
4. Naplň akumulátor z této nové adresy.

Opět si uděleme příklad:

místo	obsah
$47	$53
$48	$E4
X-registr	4

Nyní máme vykonat instrukci:

LDA ($43,X)

Přičtěme obsah registru X k základní adrese $43+4 = $47. Na paměťových buňkách $47, $48 najdeme dva byty, které interpretujeme jako novou adresu, $E453, a konečně ukončíme práci naplněním akumulátoru hodnotou z této adresy $E453.

Tato operace vyžaduje 6 cyklů a je to doposud nejpomalejší instrukce. Poněvadž používá 1 bytovou adresu, vyžaduje pouze 2 byty v paměti.

Indexované nepřímé adresování se používá poměrně zřídka. Jeho hlavní použití je vytvoření tabulky ve stránce 0 a získávání adres v paměti pomocí této tabulky. Protože však ve stránce 0 máme k disposici velmi málo místa, obecně nemůžeme tabulky do stránky 0 umístit a používáme jiných adresních módů a tabulky umisťujeme jinam.

Příkladem použití mohou být různé hry, kdy je program v paměti sám a může relativně svobodně využívat stránku 0 pro sebe.

Poznámky

Kapitola 6:

Assemblery pro ATARI

Mluvíme-li o assemblerech, mluvíme o programech, které vám dovolují psát programy v jazyce assembler a spouštět je a eventuálně ladit. Takový soubor programů se sestává obvykle ze tří částí: Editor, který slouží k vlastnímu zápisu programu; assembler, který slouží k převodu zdrojového kódu do strojového jazyka a debugger, (ladicí monitor), který nám pomůže najít chyby. Upravit je tak, aby náš produkt pracoval tak, jak jsme chtěli.

V současné době je pro ATARI k disposici několik programových souborů:

1. Assembler/Editor v zásuvném modulu (nebo jeho disketová či pásková verze) od ATARI Inc.
2. ATARI Macro Assembler(AMAC), MEDIT(Editor) a DDT(Debugger), všechny od APX,ATARI Inc.
3. EASMD od OSS
4. MAC/65, od Optimized Systems Software Inc.
5. SYNASSEMBLER, od Synapse Software
6. Macro Assembler/Text Editor (MAE) od Eastern House Software
7. Edit 6502 od LJK Enterprise
8. SIX FORKS Assembler & Linker

Zásuvný modul Assembler/Editor (ASED)

Nejprve si něco řekneme o syntaxi. ASED vyžaduje, aby každý řádek začínal číslem řádku stejně jako programy v Basicu. Každé číslo řádku je následováno alespoň jednou mezerou. Pole, která jsou na jednom řádku programu v assembleru jsou:

<číslo řádku>-<návěští>-<mnemonika>-<operand>-<poznámka>

Např. typický řádek vypadá takto:

```
10 LOOP   LDA #$0342  ;vezmi horní byte do A
```

Probereme si postupně jednu část za druhou. První pole, návěští, může nebo nemusí být uvedeno. Pokud je, můžeme na tento řádek odkazovat pomocí návěští, v tomto případě LOOP. V jiném řádku programu může provést skok na LOOP a assembler bude vědět, kam to je. Obecně se návěští používají tam, kde budeme na tento řádek odkazovat z jiné části programu. Pokud je návěští uvedeno, musí být od čísla řádku odděleno právě jednou mezerou.

První znak návěští musí být písmeno od A do Z, a ostatní znaky musí být buď znak nebo číslice mezi 0 až 9. Návěští může být 1 nebo více znaků až do délky (105 minus počet číslic v čísle řádku). Protože některé assemblery pro ATARI omezují počet znaků pro návěští, budou

všechna návěští v této knize obsahovat 6 nebo méně znaků.

Mnemonika nebo také operační kód, je instrukce procesoru 6502. V uvedeném příkladu je to instrukce LDA pro naplnění akumulátoru. Tato instrukce musí být oddělena 1 mezerou od návěští nebo 2 mezerami od čísla řádku, pokud na řádku není návěští. Např.:

```
10 LABEL LDA $0342      nebo
10  LDA $0342
```

Důvod by měl být zřejmý: Pokud by za číslem řádku byla pouze 1 mezera assembler by mnemoniku považoval za návěští a operand by se pokusil interpretovat jako mnemoniku, samozřejmě bez úspěchu.

Operand je zbytek instrukce pro 6502 a specifikuje adresu nebo číslo, se kterými bude pracovat. V tomto případě definuje operand modus absolutního adresování, v němž má být akumulátor naplněn hodnotou z paměťové buňky $ 0342. Pokud by byl operand #$24, v tomto případě by byl adresní modus bezprostřední a akumulátor by byl naplněn hexadecimálním číslem $24. Operand začíná za alespoň 1 mezerou mezi operandem a mnemonikou, i když je dovoleno více mezer. Můžete také použít tabelátor, chcete-li. V této knize budeme používat 1 mezeru.

S každou mnemonikou, která používá akumulátorový modus, operandem musí být velké písmeno A. Tudíž instrukce pro rotaci akumulátoru musí být zapsána takto.:

```
130  ROR A   ;rotuj akumulátor.
```

Pole poznámky je poslední pole na řádku a mělo by být použito k popisu činosti programu. Tzn., poznámka by neměla popisovat operaci (že LDA $0342 znamená naplnění akumulátoru z buňky $0342), spíše by vám měla později pomoci, pochopit funkci, až se po půl roce vrátíte k programu a budete několik hodin přemýšlet, k čemu tato instrukce slouží.

Poznámky mohou být od kódu odděleny dvěma způsoby. Buď je vše, co je operandem odděleno aspoň 1 mezerou, považováno za poznámku nebo je možno označit jako poznámku celý řádek. V tom případě jako první znak, oddělený od čísla řádku mezerou, napíšeme středník. Celý zbytek řádku bude považován za poznámku. Příklady obou možností.:

```
100 ;celý tento řádek je poznámka
100  LDA $0343 ;toto je rovněž poznámka
```

V této knize budou všechny poznámky uvedeny středníkem ať je na celé řádce nebo ne. Jakmile uvidíte středník před textem, víte že jde o poznámku.

Nyní známe strukturu řádku a seznámíme se s několika dalšími konvencemi.

Direktivy

Většina assemblerů má pro programátory ještě několik dalších instrukcí, které mohou být assemblerem interpretovány. Říká se jim direktivy nebo také pseudoinstrukce, protože se používají stejně jako instrukce, ale nepatří do instrukčního souboru 6502. Nejdůležitější z nich pro ASED jsou uvedeny dále s krátkým popisem.

Jeden z nejdůležitějších je příkaz ORIGIN. Poněvadž assembler vytváří kód, který bude umístěn na určitém místě v paměti potřebujeme

assembleru říci, kde toto místo je. Formát pro ASED je následující:

```
10  *= $0600  ;začátek
```

Všimněme si, že mezi číslem řádku a hvězdičkou jsou 2 mezery a mezi rovnítkem a adresou jedna mezera. Tento řádek říká assembleru, že náš program bude začínat na adrese $0600 ve stránce 6. Podobný příkaz bude obvykle prvním nebo jedním z prvních příkazů vašeho programu. Když assembler vidí příkaz *=, nastaví hodnotu programového čítače na hodnotu výrazu, který je v operandu tohoto příkazu. V programu se může takových příkazů použít několik, pokud budou různé části kódu umístěny v různých oblastech paměti.

Mezi další pseudoinstrukce patří:

.BYTE rezervuje alespoň 1 byte v paměti pro další použití. Operand může umístit do tohoto místa informaci. Např. instrukce:

```
110  .BYTE 34
```

rezervuje na okamžité pozici programového čítače jednu buňku a uloží do tohoto místa konstantu $22 (dekadické 34). Pomocí .BYTE je možno vložit i několik bytů, jak je vidět dále.:

```
125  .BYTE "HELLO",$9B
```

Tento příkaz umístí v po sobě jdoucích buňkách paměti hexadecimální čísla $48, $45, $4C, $4C, $4F a $9B. Tato čísla jsou ATASCII kódy pro písmena slova HELLO.

.DBYTE rezervuje pro každou hodnotu v operandu 2 byty. Tato pseudoinstrukce se používá pro data, v nichž je číslo větší než 256 a k uložení jsou zapotřebí 2 byty. Číslo se ukládá v pořadí horní/dolní byte. Např.:

```
115  .DBYTE 300
```

Uloží ve dvou, po sobě jdoucích bytech čísla $01,$2C, protože 300 je hexadecimálně $012C.

.WORD je stejné jako .DBYTE s tím rozdílem, že dolní byte se uloží jako první a následuje horní byte.

<návěští>= se použije pro nazvání hodnoty symbolickým jménem. Např. budeme-li v programu často používat adresu $9F, můžeme této hodnotě přiřadit pojmenování proměnné takto :

```
112 FREQ = $9F
```

<návěští> je odděleno od čísla řádku právě jednou mezerou. Kdekoliv nyní potřebujeme adresu $9F, můžeme namísto toho použít <návěští>, např.:

```
245  LDA FREQ
```

Assembler nyní ví, že má naplnit akumulátor hodnotou $9F.

.END ukončuje práci assembleru a má to tedy být poslední instrukce vašeho programu. ASED předpokládá, že pokud už nejsou další řádky kódu a nebyl uveden .END, program je ukončen, tzn. že příkaz

.END nemusí být uváděn.

Pro ovládání editace, překladu a ladění jsou k dispozici následující povely:

NUM od,přírůstek - nastavení automatického číslování řádek.Pokud napíšeme samotné NUM, začínáme řádkou 10 s přírůstkem 10. Je-li již v paměti program, pokračuje NUM s číslováním za posledním uloženým řádkem.

REN nový začátek číslování,přírůstek - instrukce sloužící k přečíslování řádek

DEL od řádku,do řádku - včetně tato instrukce nám umožní vymazat části textu mezi vyznačenými řádkami.

NEW mazaní celé textové části

SIZE dotaz na velikost paměti

FIND/řetězec/ buď od,do, nebo ,A instrukce, pomocí které vyhledáváme zadaný řetězec v určitých řádkách, nebo zadáním ,A v celém souboru. Lomítka jako oddělovače lze nahradit jiným znakem kromě mezery.Např.:

FIND-a/b-,A

REP/starý text/nový text/ buď od,do nebo ,A nebo ,Q výměna textů buď mezi zadanými řádkami, nebo ,A všude, nebo ,Q všude, ale s dotazem, zda vyměnit či ne.O oddělovačích platí totéž, co u FIND.

LIST# file spec. pomocí tohoto příkazu vypisujeme soubor na zvolené zařízení - standardně E:- vypisují se nám i čísla řádků.Takto vypsaný soubor můžeme opět načíst příkazem ENTER.

PRINT# file spec. obdobné jako u předchozího příkazu, ale nevypisuje čísla řádek.

ENTER# file spec případně ,M načtení zdrojového textu do paměti. Pokud chci spojit více souborů, napíši za specifikaci zařízení ,M.

SAVE# file spec. <začátek,konec
příkl.SAVE#C:<1F00,3000
tento příkaz slouží k uschování obsahu části paměti na příslušné zařízení

LOAD# file spec. načtení binárního souboru, uloženého příkazem SAVE.

ASM #file spec. 1,#file spec. 2,#file spec. 3
příkaz k zkompilování souboru na uvedeném zařízení.
File spec.1 - soubor, v němž je uložen zdrojový text programu.
File spec.2 - soubor, kam se uloží výpis z kompilace. Standardně obrazovka.
File spec.3 - soubor, kam se uloží výsledný binární kód. Neuvedeme-li první nebo třetí parametr, probíhá kompilace z nebo do

paměti.

DOS příkaz k přechodu do DOSu

Nyní si ukážeme jak přecházet z editoru do debugu a do DOSu.

EDIT--> BUG--> DEBUG--> X--> EDIT--> DOS--> DOS

Příkazy DEBUG:
Pozor, zde se všechna čísla uvádějí jako hexadecimální, bez znaku $.

DR zobrazí registry A,X,Y,P,S

CR <A,X,Y,P,S změna hodnot zobrazených registrů.Registry, které neměníme nahrazujeme čárkou.Např. změna reg. Y: CR<,,23

D <od,do> zobrazení paměti od - do

C adresa<č,í,s,l,a změna obsahu paměti od zadané adresy.

M nová adresa<od,do přesun bloku paměti od - do na novou adresu.

V adresa1<od,do porovná blok paměti od - do se stejně velkým blokem, začínajícím od adresy 1.

L <začátek> zobrazí instrukce v paměti od zadané adresy

G <kam> ekvivalent JMP

S <odkud> krokování programu

```
.OPT<NO LIST> tisk vypnutý
    <LIST>    tisk zapnutý
    <NO OBJ>  potlačení generování strojového kódu
    <OBJ>     povolení    -"-        -"-
    <NO ERR>  potlačení výpisu chyb
    <ERR>     povolení    -"-
    <NO EJECT>potlačení stránkování
    <EJECT>   povolení stránkování
```

.TITLE"....." titulek, který se píše jen jednou na začátku programu

.PAGE"....." dodatek k titulku, může se během programu měnit

.TAB n1,n2,n3 odsazení výpisu

Matematika v poli operandu

V poli operandu můžeme také sčítat, odečítat, dělit, násobit ap. Např. chceme-li oddělit adresu návěští LOOP na dolní a horní byte, můžeme napsat.:

```
135  LDA #LOOP&255   ;vezmi dolní byte návěští LOOP
140  STA DEST        ;ulož jej do DEST
145  LDA #LOOP/256   ;vezmi horní byte
```

```
150  STA DEST+1        ;a máme hotovo
```

Rádek 135 vezme adresu LOOP a udělá logický součin s $FF, čímž dostaneme dolní byte. Rádek 145 dělí adresu LOOP 256, čímž dostaneme horní byte. Všimněme si, že na řádku 150 uložíme tento byte na adresu DEST plus 1, a 1 byte v paměti dále než DEST.

Rozdíl mezi Assembler/Editorem a ostatními assemblery.

Macro Assembler vám umožní psát makroinstrukce, což jsou obecně části programu v Assembleru, u kterých předpokládáte časté používání v programu. Příkladem makra může být JMI, které bude obsahovat kód, implementující instrukci"SKOK PRI MINUS", která jak víme není v souboru instrukcí 6502. Když assembler při překladu narazí na jméno JMI, nahradí je posloupností instrukcí, které jsou obsazeny definicí makra JMI a realizují tuto operaci.

AMAC se liší od ostatních assemblerů svou schopností přeložit i několikrát větší soubor než celý paměťový prostor v ATARI. Čte zdrojový text ze souboru na disku, přeloží jej a cílový kód uloží zpět do jiného souboru na disku. Má další důležitou vlastnost- použití zvláštních souborů, zvaných SYSTEXT. Tyto soubory mohou obsahovat např. všechny popisy návěští, takže SYSTEXT může být napsán jednou a použit ve všech dalších programech bez nutnosti je psát vždy znovu.

Tento assembler se liší ještě v dalším aspektu - nepoužívá čísla řádků. Rádky se jednoduše vloží nebo zruší v příslušném pořadí. Návěští začíná ve sloupci 1 a mnemonika je obecně tabelátorem odsazena o cca 8 mezer. Návěští mohou mít libovolnou délku, ale pouze prvních 6 znaků je významných, delší návěští používáte na vlastní nebezpečí. Kromě binárních, dekadických a hexadecimálních čísel je možno používat i oktalová, označená pomocí znaku "%". Retězce znaků, jako HELLO užité v příkladu, jsou omezeny apostrofy a nikoliv uvozovkami. AMAC rovněž podporuje sčítání, odčítání, násobení a řadu jiných logických operací.

Adresa může být rozdělena na horní a dolní byte jednoduše použitím slov HIGH a LOW bez nutnosti dělení jako v uvedeném příkladu. Makra jsou samozřejmě dovolena. Každá instrukce v módu assembleru má mít jako operand A. Pseudoinstrukce se prakticky všechny liší od pseudoinstrukcí ASED. Jejich porovnání následuje.:

```
AMAC    ASED      poznámka

DB      .BYTE     AMAC rovněž rozumí .BYTE
DW      .WORD      - " -   - " -            .WORD
END     .END       - " -   - " -            .END
EQU     =
LOC               dosadí čítač polohy pro Assembler
ORG     *=
```

EASMD - podporuje všechny povely a pseudoinstrukce, jako ASED, má však navíc pseudoinstrukci .INCLUDE#<filespec>, pomocí níž lze rozdělit program na menší části a editovat je samostatně.

MAC/65 - dosud nejlepší assembler pro ATARI. Dolní a horní byte

16-ti bitového výrazu lze rozdělit pomocí operandů "<" a ">", např.:

```
LDA #<DLIST
LDA #>DLIST
```

V pseudoinstrukcích je praktická direktiva .SBYTE , která ukládá text v interním kódu. Stejně jako EØSMD má pseudoinstrukci .INCLUDE.

Kapitola 7:

Podprogramy pro BASIC ve strojovém kódu

Když budeme psát podprogramy, musíme se rozhodnout, kam budou umístěny. Jsou dva typy programů: přemístitelné (relokatabilní) nebo pevné. Pevné programy jsou ty, které používají uvnitř programu konkrétní adresy, které se nemohou měnit. Např. předpokládejme, že váš program obsahuje následující řádky:

```
30      *= $600
45      LDA ADDR1
50      BNE NOZERO
55      JMP ZERO
60 NOZERO RTS
70 ZERO SBC # 1
80      RTS
90 ADDR1 .BYTE 4
```

V tomto úryvku jsme použili několik odkazů na pevné adresy podprogramů; ty se nemohou změnit aniž by došlo ke zmatení programu. Lépe to uvidíme po překladu tohoto programu assemblerem. Dostaneme následující výstup:

```
ADDR   ML      LN    LABEL    00          OPERAND

0000           30             *=          $600
0600   AD0C06  45             LDA         ADDR1
0603   D003    50             BNE         NOZERO
0605   4C0906  55             JMP         ZERO
0608   60      60    NOZERO   RTS
0609   E901    70    ZERO     SBC         #1
060B   60      80             RTS
060C   04      90    ADDR1    .BYTE4
```

Tento výpis je organizován do sloupců. První sloupec uvádí hexadecimální adresy, kam budou umístěny strojové instrukce. Druhý sloupec uvádí strojový kód, který je výsledkem překladu. Např. instrukce RTS v řádku 80 generovala strojový kód $60, umístěný na adresu $060B. Třetí sloupec uvádí čísla řádek zdrojového programu. Čtvrtý sloupec obsahuje operand. V tomto příkladu není sedmý sloupec, který by obsahoval poznámky z původního programu.

Vrátíme se ale k problému pevných adres. První z nich ADDR1 je použita v řádcích 45 a 50. Prohlédněme si kód, vygenerovaný assemblerem v řádku 45. Jsou to tři byty: AD, 0C, 06. AD je strojový kód pro LDA v absolutním adresním modu. Druhý a třetí byte 0C a 06 tvoří adresu, v tom případě $060C. Nyní je jasné, proč tento program nemůže běžet v jiné oblasti paměti. Při provádění LDA na řádku 45 bude procesor plnit akumulátor podle původní adresy $060C, protože tam bylo ADDR1 v době překladu.

Druhá pevná adresa v tomto programu je v řádku 55. Každá instrukce JMP má jako cílovou adresu pevnou adresu, v tomto případě $0609. Pokud bychom opět program přesunuli někam jinam, v okamžiku skoku se program zhroutí.

Všimněme si na chvíli řádku 50. Víme že všechny větvící instrukce používají relativní modus adresování. Strojový kód pro tuto instrukci je D0,03. D0 je kód pro BNE, ale 03 není adresa, je tím pouze vyjádřeno, že cílová adresa je o 3 byty dále, v tomto případě na řádku 60. Protože větvící instrukce říká, "skoč o 3 byty" a ne "skoč na adresu $0608", může být umístěna kdekoliv v paměti a podrží si svůj význam.

POZOR: Z toho plyne, že v přemístitelném kódu mohou být instrukce větvení, ale instrukce JMP a absolutní adresy nikoliv!

Proč mluvíme tolik o přemístitelném kódu? Jednoduše proto : pokud kód není přemístitelný, musíme v paměti počítače najít nějaké bezpečné místo pro jeho umístění. To není vždy snadné, protože sdílíme paměť počítače s BASICem a nevíme vždy s jistotou které buňky paměti BASIC používá. Jestliže je náš kód přemístitelný, můžeme jej dát kamkoliv. A kam?

Podívejme se jak pracuje BASIC se znakovými řetězci. Chceme-li použít řetězec v ATARI BASICu, musí být pro něj vyhrazeno místo v paměti. Jestliže z nějakého důvodu potřebuje část této oblasti, přesune jej někam jinam, ale BASIC je přitom zodpovědný za, to aby si zapamatoval, kde řetězec je, a rovněž za ochranu tohoto místa před přepsáním. Můžete tedy umístit váš program jako řetězec v BASICu a vyvolávat jej pomocí USR (ADR(řetězec $)).

Pro přesnost dodejme, že vždy bude nějaké místo pro krátký podprogram ve stránce 6, která je zaručeně volná pro použití programátorem. Téměř vždy. Za určitých, málo pravděpodobných podmínek nemusí být stránka 6 zcela bezpečná. Jak bylo řečeno v kap.3, místo od $ 580 do $ 5FF (horní polovina 5 stránky) se používá jako vyrovnávací paměť. Zadáte-li informaci z klávesnice, tento vstupní buffer může přetéci až na začátek stránky 6. Toto přetečení pak přepíše cokoliv mezi $ 600 a $ 6FF v závislosti na své velikosti. Pro účely této knihy budeme předpokládat, že k tomuto přetečení nedojde, a programy, které mohou být napsány jako nepřemístitelné, budou obecně začínat na $ 600. Pokud by někdy nechtěly pracovat, ujistěte se, že nedochází k přetečení na stránce 5.

Jiná místa pro nepřemístitelné programy jsou nahoře v paměti nebo pod LOMEM. Obě místa jsou obecně bezpečná před BASICem, pokud jsou používána s jistou obezřetností. Navíc, pokud máte aplikaci, která NIKDY nepoužije msf. pásku, můžete použít buffer pro msf., umístěný mezi $ 480 a $ 4FF. Velmi krátké podprogramy mohou být také umístěny na začátku zásobníkové paměti ve stránce 1 od $ 100 asi k $ 150, poněvadž jen málokterý program použije zásobníkovou paměť do těchto buněk. Toto je však zvláště riskantní a nikdo nezaručí spolehlivou činost podprogramů využívajících toto místo.

Mluvíme-li nyní o magnetofonech, je nutno udělat ještě jednu poznámku. Všechny programy zde předpokládají použití disketové jednotky a rezidentní DOS. Používáte-li systém, založený na magnetických páscích, zjistěte si v manuálu k vašemu assembleru jak provádět některé operace. Např. k načítání souborů ve strojovém jazyce z disku může využívat funkci L v DOSu, zatímco k téže operaci z magnetofonu budeme pravděpodobně potřebovat některou z instrukcí LOAD nebo BLOAD v závislosti na použitém assembleru.

Jednoduchý příklad.

Podprogram pro nulování paměti.

Začneme si budovat knihovnu vlastních podprogramů velmi jednoduchým příkladem.

V BASICU často potřebujeme vynulovat určitou oblast paměti. To se stává např. při používání PMG <player-missile graphic> nebo při použití paměti jako zápisníku ap. Připomeňme si, že chceme-li uchovat data v horní části paměti, musí být display-list a videopamet přemístěna pod tuto část, jinak bychom si takto snadno vymazali obsah obrazovky. Toto přemístění je snadno možné pomocí následujícího programu v BASICU.:

```
10 ORIG=PEEK(106):REM uchovej původní vršek paměti
20 POKE 106,ORIG-8:REM sniž vršek paměti o 8 stránek
30 GRAPHICS 0:REM přesuň display-list a display-memory
40 POKE 106,ORIG:REM vrat zpět původní vršek paměti
```

Samozřejmě bychom mohli napsat podprogram pro nulování paměti v BASICu následujícím programem.:

```
10 TOP=PEEK(106):REM najdi vršek paměti
20 START=TOP-8)*256:REM spočti kde začít s nulováním
30 FOR I=START TO START+2048:REM nulovaná oblast
40 POKE I,0:REM vynuluj každou buňku
50 NEXT I:REM končíme
```

Tento program pracuje tak, jak bychom chtěli, ale trvá mu to cca 13 sekund. Potřebujeme-li tuto operaci ve svém programu několikrát nebo je-li to pro nás příliš dlouhá doba, máme dobrého kandidáta na podprogram.

Nejdříve musíme pro něj najít místo, např. ve stránce 6. Buňka 106 vždy obsahuje informaci o tom kde je konec paměti. Pro tento typ programu je obvykle výhodné použití nepřímého idexového modu, takže budeme potřebovat dvě buňky ve stránce 0 pro nepřímou adresu. Zapišme to, co teď víme, v assembleru.:

```
100 ;xxxx
110 ;nastavení počátečních podmínek
120 ;xxxx
130     *= $600; začátek programu na stránce 6
140 TOP= 106; tady najdeme konec paměti
150 CURPAG =$CD; sem umístíme adresu nulované stránky
```

Na stránce 0 jsou pouze 4 adresy $CC až $CF, které jsou bezpečné před příkazy BASICu nebo ASEDu. Použijeme dvě z nich $CC a $CD. Dají se najít i jiné volné buňky, ale tyto čtyři jsou zaručeně volné, použijeme tedy tyto.

Nyní se soustředíme na vlastní podprogram. Jako první musíme použít PLA pro odstranění počtu parametrů ze zásobníkové paměti, kam jej vložil BASIC při volání USR. Poté, co zjistíme současný konec

paměti, začneme o 8 stránek níže s nulováním paměti. Vyhledání místa v paměti, kde začít s nulováním, zapíšeme následovně:

```
160 ;xxxx
170 ; začneme výpočtem
180 ;xxxx
190     PLA ; odstraň počet parametrů ze zásobníku
200     LDA TOP ; najdi konec paměti
210     SEC     ; příprava na odečítání
220     SBC #8  ; najdi první nulovanou stránku
230     STA CURPAG ; poznamenej si ji
240     LDA #0
250     STA CURPAG-1 ; dolní byte adresy je nula
```

Nyní máme připravenu nepřímou adresu první nulované buňky v paměti v CURPAG-1<dolní byte> a CURPAG <horní byte>. Zároveň máme v akumulátoru nulu , kterou budeme ukládat do buněk paměti.

Dále potřebujeme čítač pro hlídání počtu nulovaných buněk v každé stránce. Použijeme-li registr Y, bude nám sloužit zároveň jako čítač i jako index. Ještě musíme nastavit čítač, uložit nulu z akumulátoru do paměti, zmenšit Y o 1 a vrátit se zpět. Poněvadž začínáme s 0 v čítači a pokaždé snižujeme jeho hodnotu o 1, budeme pokračovat tak dlouho dokud v Y znova nebude 0. Tím zajistíme vynulování všech 256 buněk na stránce:

```
260  LDY #0 ; použití Y jako čítače
270 ; xxxx
280 ; následuje nulovací smyčka
290 ; xxxx
300 LOOP STA <CURPAG-1>,Y ; vlastní nulování buňky
310  DEY ; snížení čítače
320  BNE LOOP ; dokud <>0, pokračuj
```

Tím jsme vynulovali první stránku. Jak vynulovat zbývajících 7. Vzpomeňme si, že nepřímá adresa ve stránce 0 má 2 byty<dolní a horní>. Zvětšíme-li horní byte o 1, tato nepřímá adresa bude ukazovat na nejblíže vyšší stránku:

```
330 INC CURPAG ; ukazuj na další stránku
```

Nyní musíme ověřit, zda ještě pokračovat. Víme že musíme skončit, když číslo nulované stránky překročí konec paměti:

```
340  LDX CURPAG ; potřebujeme pro porovnání
350  CPX TOP    ; porovnej s koncem paměti
360  BEQ LOOP   ; jdi nulovat poslední stránku
370  BCC LOOP   ; ještě je třeba nulovat
380  RTS        ; návrat do BASICu
```

Pokud je CURPAG=TOP, stále ještě musíme vynulovat poslední stránku. Teprve když CURPAG>TOP, končíme s nulováním.

Když jsme nyní napsali náš program, vyvoláme Assembler, což v ASED uděláme napsáním povelu ASM, následovaným RETURN.

Tím se náš program přeloží a uloží do paměti. Na monitoru se vám zároveň objeví výpis z překladu. Dále musíme program uložit na disk tak abychom jej mohli použít v programu v BASICu. To můžeme udělat dvěma způsoby. První možnost je přímo z ASED příkazem SAVE:

```
SAVE #D :PROGRAM<0600,061F
```

Tento příkaz vytvoří na disku soubor se jménem PROGRAM a uloží do něj obsah paměti od $600 do $061F.

Druhá možnost je přejít do DOS a použít příkaz K (v DOS 2,5) s parametry PROGRAM,0600,061F.

Nyní můžeme nastartovat BASIC. Po výpisu READY přejdeme do DOSu příkazem DOS a použitím příkazu L načteme ze souboru PROGRAM na stránku 6 náš podprogram. Příkazem B pak přejdeme znovu do BASICu.

Náš podprogram je nyní ve stránce 6 a chceme-li, můžeme jej používat. Dalším krokem by však měla být transformace tohoto programu do takové formy, abychom nemuseli pro načítání používat DOS.

Použijeme univerzální program který přečte data z paměti a upraví jej do formy, ze které se snadno převede do programu v BASICu.:

```
 5 BASNO= 10000:REM číslo řádku v BASICu pro DATA
10 FOR J=1 to 30 STEP 10:REM délka dat v paměti
15 PRINT BASNO;" DATA ";:REM začátek řádku
16 BASNO=BASNO+10:REM zvětši číslo řádku
20 FOR I=J TO J+9:REM 10 bytů na řádek
30 PRINT PEEK(I+1535);",";:REM vypiš data na monitor
40 NEXT I:REM ukonči smyčku
50 PRINT:REM ukonči buňku
60 NEXT J:REM konec
```

Jestliže nyní napíšeme tento program, a spustíme jej, na monitoru se objeví.:

```
10000 DATA 104,165,.....
10010 DATA 133,....
10029 DATA 205....
```

Přemístíme kursor na tyto řádky a zavedeme je do programu. Nyní můžeme zrušit řádky 5 až 60 a program v paměti se bude sestávat pouze z řádek 10000 až 10020 a v tomto okamžiku jej můžeme pomocí LIST uchovat na disku k příštímu použití.

Takto vytvořený podprogram má jednu nevýhodu- je vytvořený na stránce 6 a nemůže být použit s programem, který potřebuje stránku 6 pro sebe.

Nyní bychom mohli využít toho, že náš program neobsahuje absolutní adresy a je proto přemístitelný. Převést data do řetězce můžeme např. pomocí tohoto programu v BASICu.:

```
 5 DIM CLEAR$(30):REM vytvoř řetězce
10 FOR I=1 TO 30:REM délka řetězce
20 READ A:REM přečti jednotlivé byty
30 CLEAR$(I,I)=CHR$(A):REM přenos byte do řetězce
40 NEXT I
 :
100 X=USR(ADR(CLEAR$)):REM vyvolání podprogramu
 :
10000 DATA
10010 DATA
10020 DATA
```

Další možností je takto vytvořený řetězec vypsat na obrazovku a převést jej na jediný řádek v BASICu.:

```
20000 E=USR(ADR(".............")):RETURN
```

Musíme však přitom dát pozor aby náš řetězec neobsahoval řídící znaky, nejlépe pomocí kontroly, zda počet znaků odpovídá původnímu počtu bytů. Chybějící byty můžeme doplnit s využitím znaku ESC, ale u delších řetězců může být tato práce dosti zdlouhavá a vyplatí se pouze tam, kde nám opravdu jde o úsporu místa nebo času. V opačném případě plně postačí načítání z příkazu DATA.

Podprogram pro přesun části paměti

Jako další příklad podprogramu ve strojovém jazyku uvedeme přesun několika stránek paměti. Při volání podprogramu musíme zadat parametry ODKUD,KAM,POCET, kde POCET je počet stránek, které se mají přesunout; stále předpokládáme, že přesouváme celé stránky. Program v assembleru následuje.

```
100 ;xxx
110 ;dosaď počáteční podmínky
120 ;xxx
130     *=$600
140 ODKUD = $CC
150 KAM = $CE
160 ;xxxx
170 ;inicializace , přenos parametrů
180 ;xxxx
190         PLA             ;počet parametrů ze zásobníku
200         PLA             ;horní byte 1 parametru
210         STA ODKUD+1
220         PLA             ;dolní byte 1 parametru
230         STA ODKUD
240         PLA             ;horní byte 2 parametru
250         STA KAM+1
260         PLA             ;dolní byte 2 parametru
270         STA KAM

280         PLA             ;horní byte 3 parametru (=0)
290         PLA             ;dolní byte 3 parametru = počet
                             stránek
300         TAX             ;počet stránek do čítače X
310 ;xxx
320 ;přenos paměti
330 ;xxx
340         LDY #0          ;inicializace čítače bytů
350 ZNOVU   LDA (ODKUD),Y;vezmi byte
360         STA (KAM),Y     ;ulož byte
370         DEY             ;je už konec strany ?
380         BNE ZNOVU       ;ještě ne - opakuj
390         INC ODKUD+1     ;zvětši čítač stránek ODKUD
400         INC KAM+1       ;zvětši čítač stránek KAM
410         DEX             ;všechny stránky přesunuty ?
420         BNE ZNOVU       ;ne,opakuj
430         RST             ;návrat do BASICu
```

Program v BASICu , který použijeme k přenosu znakové sady z ROM do RAM .

```
10 GOSUB 20000: REM vytvoř strojový podprogram
```

```
20 ORIG =PEEK(106): REM konec RAM
30 CHSET =(ORIG-4)*256: REM nové uložení znakové sady
40 POKE 106,ORIG-8: REM udělej pro ni místo
50 GRAPHICS 0:REM vytvoř nový display list
60 x=USR (ADR(TRANSFER$),57344,CHSET,4) :REM přenes celou
   sadu
70 END
20000 DIM TRANSFER$ (33):REM vytvoř program jako řetězec
20010 FOR I=1 TO 33 :REM naplň řetězec
20020 READ A :REM vezmi byte
20030 TRANSFER$(I,I)=CHR$(A) :REM ulož do řetězce
20040 NEXT I :REM opakuj
20050 RETURN :REM hotovo vrat se
20060 DATA
20070 DATA
20080 DATA
20090 DATA
```

Kapitola 8:

Display-list a použití přerušení

Čip ANTIC

Váš počítač ATARI se v jednom ohledu liší od ostatních dostupných domácích počítačů. Většina z nich obsahuje jeden mikroprocesor, 6502 nebo Z80 nebo některý jiný. Váš počítač ATARI má mikroprocesory 4, z nich tři byly vyvinuty speciálně pro ATARI a jsou osazeny pouze v něm. V této části si povíme o jednom z nich, který se nazývá ANTIC.

V počítači ATARI řídí zobrazení na obrazovce čip ANTIC, což je důležitá vlastnost počítačů ATARI. V jiných mikropočítačích řídí generaci videa hlavní mikroprocesor vedle vlastních výpočtů. Firma ATARI Corp. vyvinula čip ANTIC, aby odlehčila 6502 od této zátěže, takže ANTIC řídí zobrazení a 6502 vlastní výpočty.

Videopaměť

Část paměti RAM je vyhrazena pro informaci, která je zobrazována na obrazovce televizoru nebo monitoru. Tuto oblast paměti označujeme jako obrazová nebo video pamět. Jako většina vyhrazených oblastí paměti použitých pro speciální účely má i videopaměť svůj ukazatel, který nám vždy řekne, kde je videopaměť umístěna, i když ji přeneseme jinam. Tento ukazatel je umístěn ve dvou bytech na adresách 88 a 89. Obecně vzato, kdykoliv použijete příkaz GRAPHICS X, operační systém vytvoří videopaměť bezprostředně před koncem paměti. Protože velikost paměti, potřebná pro videopaměť se může velice měnit v závislosti na zvoleném modu zobrazení, je důležité vědět, kde videopaměť začíná, a na uvedené adrese se to dozvíme. Pro určení začátku videopaměti stačí jediný řádek v BASICu:

```
10 BEGDM=PEEK(88)#256 PEEK(89)
```

Podívejme se, k čemu tuto informaci můžeme využít.

Víme, že použitím příkazu GRAPHICS 0 v BASICu se obrazovka vyprázdní. Ve skutečnosti operační systém zjistí podle obsahu buňky 106 konec paměti. Pak určí velikost videopaměti pro zvolený zobrazovací modus a automaticky vynuluje tuto oblast paměti, takže obrazovka bude prázdná. Nakonec se vytvoří display-list, o němž se také zmíníme, a řízení se předá BASICu.

Jakmile jsme nastavili zobrazení do modu GRAPHICS 0, víme, že můžeme zobrazit písmeno A v levém horním rohu obrazovky napsáním

```
PRINT "A"
```

Totéž můžeme dosáhnout i jinak. Nyní, když víme, kde je videopaměť umístěna v paměti RAM, můžeme příslušnou hodnotu pro A dát do paměti příkazem POKE a písmeno se objeví na obrazovce stejně jako

kdybychom použili PRINT.

```
POKE BEGDM+2,33
```

Zvýšení +2 v tomto příkaze odpovídá tomu, že zobrazení u ATARI standardně začíná ve 2. sloupci. 33 odpovídá písmenu A v příslušném kódu. Všimněme si, že počítač ATARI rozlišuje tři rozdílné sady kódů pro význam všech 256 možných kombinací jenoho bytu. První z nich je kód ATASCII nebo ATARI ASCII, který se používá např. v BASICU:

```
PRINT CHR$(65)
```

Tímto příkazem se na stínítko napíše A. Druhý z kódů je interní displejová sada, ve které písmenu A odpovídá kod 33, jak bylo uvedeno výše. Tento kód použijeme při přímém ukládání informace do videopaměti. Třetí sada se jmenuje klávesový kód , který se použije při čtení kláves z klávesnice. Nejobvyklejší použití je při zjišťování klávesy, která byla stisknuta jako poslední. Pokud adresa 764 obsahuje 255, nebyla stisknuta žádná klávesa. Při čekání na stisk klávesy můžeme napsat:

```
100 POKE 764,255
110 IF PEEK(764)=255 THEN 110
```

Chceme-li se dozvědět, která klávesa byla stisknuta, musíme použít klávesový kód. Jestliže např. PEEK(764)=127, bylo naposledy stisknuto velké písmeno A.

Tyto tři znakové sady jsou uvedeny v příloze 2. Veškeré výpisy na obrazovku můžeme zajistit pomocí vnitřního kódu pomocí POKE příslušné informace do správného místa ve videopaměti, tak jak jsme to udělali s písmenem A výše.

Udělejme si pokus. Umístíme stejný kód 33 do videopaměti, ale namísto GRAPHICS 0 Použijeme jiný grafický modus.

```
10 FOR MODE=0 TO 8:REM Gr. mody
20 GRAPHICS MODE:REM dosaď modus
30 BEGDM=PEEK(88)+256*PEEK(89):REM Kde je videopaměť
40 POKE BEGDM+2,33:REM zobraz A
50 FOR DELAY=1 TO 700:NEXT DELAY: REM umožni prohlédnout
   zobrazení
60 NEXT MODE:REM nový modus
```

Při běhu tohoto programu uvidíme, že dochází k něčemu velmi zajímavému. Nejprve v módu 0 se předpokládané A objeví v levém horním rohu. V GRAPHICS 1 a 2 se objeví středně velké a velké žluté písmeno A. Avšak v dalších grafických modech se neobjeví písmeno a vidíme pouze body různých barev.

Display list

Důvodem pro tyto rozdíly mezi grafickými mody je způsob, jakým ANTIC interpretuje obsah videopaměti. Pokud by ANTIC pouze vzal obsah videopaměti a přenesl jej na obrazovku, neušetřil by procesoru 6502 moc práce. Procesor 6502 musí pouze připravit krátký program pro ANTIC, kterým je mu zároveň řečeno, jak si 6502 přeje videopaměť interpretovat, a ANTIC zařídí zbytek. Tento program se nazývá display-list. Abychom si plně uvědomili všechny možnosti, které dává display-list programátorovi, musíme se naučit další programovací

jazyk. Naštěstí nemá příliš mnoho instrukcí, takže naučit se mu je velice snadné.

Uvedeme si nyní jeho instrukce v dekadickém i hexadecimálním tvaru a pak si popíšeme každou instrukci podrobněji.

Hex	Dek	Instrukce
0	0	Vynechej 1 volný řádek
10	16	Vynechej 2 volné řádky
20	32	Vynechej 3 volné řádky
30	48	Vynechej 4 volné řádky
40	64	Vynechej 5 volných řádků
50	80	Vynechej 6 volných řádků
60	96	Vynechej 7 volných řádků
70	112	Vynechej 8 volných řádků
2	2	Zobraz jako v modu GRAPHICS 0
3	3	Zobraz ve speciálním text. modu
4	4	Zobraz ve 4-barev. text. modu
5	5	Zobraz ve 4-bar.velkém text.m.
6	6	Zobraz jako GRAPHICS 1
7	7	Zobraz jako GRAPHICS 2
8	8	Zobraz jako GRAPHICS 3
9	9	Zobraz jako GRAPHICS 4
A	10	Zobraz jako GRAPHICS 5
B	11	Zobraz jako GRAPHICS 6
C	12	Zobraz spec. 160x20,2-bar.gr.m.
D	13	Zobraz jako GRAPHICS 7
E	14	Zobraz spec. 160x40,4-bar.gr.m.
F	15	Zobraz jako GRAPHICS 8
1	1	Skok na adresu, uvedenou v následujících 2 bytech
41	65	Skok na adresu, uvedenou v následujících 2 bytech a čekej na vertikální zatemňovací impuls

Dále mohou být použity další 4 povely, dosadíme-li některý ze 4 horních bitů instrukce do 1. Jsou to:

Bit	Instrukce
4	Povolení jemného vertikálního posuvu
5	Povolení jemného horizontálního posuvu
6	Následující 2 byty ukazují na videopaměť
7	Další řádek způsobí přerušení od display-listu.

Zdá se to být najednou mnoho, ale probereme-li si instrukce krok za krokem, bude to docela snadné. Začneme tím, že si jednoduchý display-list prohlédneme. To se dá udělat snadno, protože podobně jako videopaměť má i display-list svůj ukazatel, umístěný na buňkách 560 a 561. Napíšeme krátký program v BASICu k vytištění display-listu, abychom si jej mohli prohlédnout.

```
10 GRAPHICS 0:REM Jednoduchý display-list
20 DL=PEEK(560)+256*PEEK(561):REM Adresa diplay-listu
30 FOR I=DL TO DL+31:REM Délka display-listu
40 PRINT PEEK(I);" ";:REM Za každým bytem vynechej mezeru
50 NEXT I
```

Po spuštění tohoto programu se nám vypíše:

112 112 112 66 64 156 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 65 32 156

Je-li paměť vašeho počítače menší než 48K, mohou se poslední dva byty a pátý a šestý byte poněkud lišit. Proberme si tento display-list byte po bytu a uvědomme si zároveň, že je to program pro procesor, kterým je v tomto případě ANTIC. Podíváme-li se do seznamu instrukcí, uvedeného výše, uvidíme, že 112 znamená vynechání 8 obrazových linek. Protože je tam instrukce 112 třikrát, mohlo by to znamenat, že na začátku se má vynechat 24 obrazových linek.

Většina televizorů je navržena tak, že na stínítku se ve skutečnosti nezobrazí celý přenášený obraz. Můžete si všimnout, že na některém televizoru začíná výstup z vašeho počítače blíže k hornímu okraji nebo blíže k levé nebo pravé straně stínítka než na jiném. Aby nedošlo ke ztrátě textu na žádném televizoru, začíná většina display-listů právě 24 volnými linkami. Musíme si ovšem uvědomit, že v módu GRAPHICS 0 je každý znak vysoký 8 bitů. Je tudíž 24 volných linek ekvivalentních místu, které by zabraly tři řádky textu . Normální stínítko v modu GRAPHICS 0 obsahuje 192 linek (8x24). Můžete přidat řádek textu nebo dva vlastní úpravou display-listu, ale i když to může vypadat hezky na vašem televizoru, nemusí to platit i pro televizor někoho jiného.

Další tři byty v display-listu byly 66, 64 a 156. Podíváme-li se do seznamu instrukcí pro ANTIC, 66 tam vůbec neuvidíme. Tato 66 je součet 64+2. 64 je odvozeno od dosazeného bitu 6 v instrukci 2 pro ANTIC. Tento byte říká ANTIC, že chceme zobrazit řádek textu v modu GRAPHICS 0 a protože je současně dosazen bit 6, znamená to, že následující 2 byty obsahují ukazatel, kde se nachází videopaměť pro tento a všechny další řádky, dokud se nenarazí na další obdobnou instrukci. Tyto 2 byty (64,156) jsou v typickém pořadí dolní-horní byte. Protože 64+256*156 = 40000, víme, stejně jako ví ANTIC, že videopaměť je umístěna na adrese 40000 a dále. Dalších 23 bytů display-listu jsou samé 2 a jednoduše sdělují ANTICu, že požadujeme všechny řádky v modu GRAPHICS 0. Společně s první instrukcí to dává dohromady 24 řádků t.zn. obvyklé stínítko v modu GRAPHICS 0. Další instrukce display-listu je 65, která znamená skok s čekáním na vertikální zatemňovací impuls.

Vertikální zatemňovací impuls

Abychom správně pochopili činnost programu, musíme si nejprve něco povědět o tom, jak vzniká obraz na stínítku televizoru. Vnitřní strana přední stěny obrazovky je pokryta látkou, která světélkuje při ozáření elektronovým paprskem. V hrdle obrazovky je katoda, která vyzařuje elektrony. Horizontální a vertikální poloha, ve které proud elektronů zasáhne stínítko, je řízena vychylováním paprsku přesně definovaným způsobem. Z pohledu osoby, pozorující obrazovku, začíná paprsek svou dráhu v levém horním rohu obrazovky a nakreslí jednu linku až k pravému okraji. Pak přeskočí zpět na linku 2 a tak dále. Intenzita paprsku se při pohybu mění, čímž vznikají světlejší a tmavší

body, které vytvářejí obraz. Zařízením tohoto druhu se říká zařízení s rozmítaným obrazem (raster scan) a jsou to nejčastější zařízení pro vytváření elektronických obrazů. Druhý hlavní typ zařízení je vektorové zařízení, v němž paprsek elektronů kreslí čáru od jejího začátku do konce. Zařízení s rozmítaným obrazem nakreslí čáru tak, že přitom paprsek proběhne celé stínítko. Ve vektorovém zařízení paprsek proběhne pouze zobrazovanou čáru.

Poté co paprsek u zařízení s rozmítaným obrazem proběhne celé stínítko, vrátí se do levého horního rohu a čeká na synchronizační impuls, který dá signál k zahájení dalšího snímku (frame). Tuto pauzu můžeme vidět na přijímači, když nastavíme vertikální synchronizaci do polohy, kdy obrázek začíná svisle popojíždět. Široký horizontální pruh, který se pohybuje ve svislém směru přes obrazovku, vzniká tím, že paprsek čeká na synchronizační signál. Tomuto intervalu, po který se paprsek elektronů nepohybuje po stínítku, se říká vertikální zatemňovací impuls.

Váš počítač ATARI vytváří 292 vodorovných linek v každém snímku a obsah obrazovky se úplně vymění 50 krát za sekundu. To se zdá být velká rychlost, ale ve srovnání s rychlostí počítače probíhá kreslení obrázku hlemýždí rychlostí. Nakreslení celého obsahu obrazovky trvá 16768 mikrosekund a zatemňovací impuls trvá okolo 1 400 mikrosekund. Uvědomíme-li si, že 1 strojový cyklus počítače je kratší než 1 mikrosekunda, bude nám zřejmý poměr rychlostí počítače a televizoru.

Instrukce skoku a čekání na vertikální zatemňovací impuls, s níž jsme se setkali výše, je ve skutečnosti 3 bytová instrukce. Říká ANTICu, že následující instrukce je na adrese, uvedené v následujících 2 bytech, v tomto případě 32 a 156. To je zároveň počáteční adresa display-listu - stejná jako ta, kterou jsme našli v buňkách 560 a 561, o nichž jsme mluvili výše. Instrukce skoku a čekání zároveň říká ANTICu, aby s pokračováním počkal až do skončení vertikálního zatemňovacího impulsu. Toto čekání zajistí dvě věci. Jednak synchronizuje počítač a televizor tak, že obraz je stabilní a dále dává 50 krát za sekundu počítači k dispozici kolem 1400 mikrosekund, kdy se nic jiného neděje. ATARI využívá tento čas pro interní záležitosti jako opravy času a spoustu dalších. O jiném využití této doby si povíme později.

Rozlišovací schopnost obrazu

Poslední poznámka o televizním obraze: i když je každý snímek tvořen 262 linkami, na většině obrazovek je vidět pouze 192, takže největší rozlišovací schopnost ATARI je 192 bodů (pixels) vertikálně. Ve vodorovném směru je nejvyšší využitelná rozlišovací schopnost 160 bodů, i když GRAPHICS 8 využívá 320. V módu GRAPHICS 8 však všichni víme, jaký zmatek v barvách to působí. Jestliže nakreslíme na obrazovce v módu GRAPHICS 8 diagonální čáru, čára se jeví v různých barvách v závislosti na umístění na obrazovce. K docílení skutečných barev, musí být rozsvíceny 2 sousední body, jinak se může objevit pouze jedna ze základních barev televizoru tam, kde jsme chtěli zobrazit bílou. Je-li důležité věrné zobrazení barev, je horizontální rozlišovací schopnost limitována na 160 bodů.

Přímý přístup do paměti

Nyní začínáme chápat, jak ATARI vytváří obraz. Shrneme: část paměti (videopaměť) je použita k uložení informace, která se má zobrazit, a tato informace je interpretována ANTICem s použitím programu, nazývaného display-list. K tomu ještě další poznámku: ANTIC a 6502 sdílejí oblast paměti RAM, zvanou videopaměť. 6502 vytváří a mění tuto informaci, ANTIC ji čte, interpretuje a zobrazuje na stínítku. Je zřejmé, že oba mikroprocesory nemohou číst paměť současně. Pokud paměť potřebuje procesor 6502, ANTIC ji nemůže číst a v době, kdy videopaměť čte ANTIC, je 6502 vypojen. Čtení paměti pomocí ANTIC nazýváme přímý přístup do paměti (DMA). Přitom ANTIC "krade" čas 6502 a během této doby 6502 nepracuje. Jakmile ANTIC skončí čtení, 6502 pokračuje v práci. Tento proces poněkud zpomaluje práci a program může být zákazem DMA urychlen asi o 30 procent. Pro zastavení DMA z BASICu jednoduše provedeme:

```
POKE 559,0
```

Pro znovuspuštění DMA:

```
POKE 559,34
```

Toto urychlení má jednu vážnou nevýhodu, obrazovka ztemní až do doby, kdy DMA znovu spustíme. Cokoliv, co v této době napíšeme na stínítko se ale objeví, jakmile povolíme DMA.

Když teď víme, jak je televizní obraz vyráběn, můžeme začít s jeho modifikacemi. O tom, jak vytvořit vlastní display-list, např. kombinací různých textových modů případně i grafiky byla již zveřejněna řada článků. Zbytek této kapitoly bude věnován programům, které nemohou být napsány v BASICu, ale které mohou být z BASICu vyvolány s použitím podprogramů ve strojovém kódu. Použijeme je k různým zajímavým účelům.

Zpracování přerušení

V kontextu této knihy budeme za přerušení považovat signál, který říká 6502, aby přerušil práci ať je v kterémkoliv místě a namísto toho udělal něco jiného, co určí programátor. Po skončení této úlohy může 6502 pokračovat v práci rozdělané před přerušením. Obvykle se používají dva typy přerušení, které oba souvisí s vytvářením televizního obrazu - přerušení z display-listu a přerušení při vertikálním zatemnění. Ani jeden z nich nemůže být použit bez podprogramů ve strojovém kódu, protože jazyky jako BASIC jsou příliš pomalé pro podobné účely.

Přerušení z display-listu

Nejprve si řekneme o přerušení z display-listu. Když jsme mluvili o display-listu, uvedli jsme, že při dosazení bitu 7 (horní bit) je vybuzeno přerušení z display-listu pro následující linku na stínítku. Co to znamená:

Na stránce 2 paměti RAM v buňkách $200 a $201 (dekadicky 512 a 513) je vektor přerušení od display-listu. Jak jsme si již řekli, je vektor něco jako ukazatel, který někam ukazuje. Normálně buňka $200 obsahuje $B3 a buňka $201 obsahuje $E7, takže tento ukazatel ukazuje na $E7B3. Tato buňka obsahuje $40, což je strojová instrukce RTI,

návrat z přerušení. Jinými slovy: vektor přerušení od display-listu normálně ukazuje na konec programu obsluhy přerušení. Je to proto, aby v případě, že povolíte přerušení od display-listu, neskočil počítač na nějakou náhodnou adresu.

Pro používání přerušení od display-listu jsou důležité úvahy o době trvání obslužného programu. Normálně tento obslužný program sestává ze tří částí. První část probíhá v době, kdy elektronový paprsek končí kreslení linky, v níž byl dosazen bit 7. Druhá část nastane v době mezi začátkem nové linky a vstupem paprsku do viditelné části linky. Třetí část začíná se vstupem paprsku do viditelné oblasti stínítka a končí na konci obslužného programu přerušení.

Elektronový paprsek potřebuje k přeběhnutí stínítka 114 strojových cyklů. I když je bit 7 dosazen na začátku linky, 6502 se o přerušení dozví až po cyklu 35. Je tedy zřejmé, že zde nemohou být implementovány dlouhé strojové podprogramy; není zde pro ně dost času.

Jednoduchý příklad

Přerušení z display-listu se všeobecně používá ke změně barvy pozadí uprostřed obrazovky. Napíšme a implementujme si takový podprogram. 6502 přerušujeme v době, kdy provádí instrukce. Proto chceme-li použít některý registr nebo akumulátor, musíme jejich obsah předem uschovat a před návratem z obsluhy přerušení opět obnovit. Prohlédněme si program a pak si o něm něco řekneme:

```
                0100 ; ********************
                0110 ; příprava
                0120 ; ********************
0000            0130 ;      *= $600 ;začátek
D40A            0140 WSYNC  = $D40A
D018            0150 COLPF2 = $D018 ;pozadí
                0160 ; ********************
                0170 ; vlastní podprogram
                0180 ; ********************
0600 48         0190        PHA         ;uchovej akumulator
0601 A942       0200        LDA #$42    ;kmitočet červené barvy
0603 8D0AD4     0210        STA WSYNC   ;viz výklad
0606 8D18D0     0220        STA COLPF2  ;dosaď novou barvu
                0230 ; ********************
                0240 ; obnov akumulator
                0250 ; ********************
0609 68         0260        PLA         ;obnova
060A 40         0270        RTI         ;končíme
```

První, čeho si všimneme v tomto podprogramu je, že nezačíná instrukcí PLA. Jediná PLA instrukce v programu je použita pro obnovení původní hodnoty v akumulátoru, tato hodnota tam byla uložena v řádku 190 pro bezpečnou úschovu v době provádění tohoto podprogramu. Tento podprogram je míněn pro spolupráci s BASICem a víme, že každé volání USR z BASICu vyžaduje PLA pro odstranění počtu parametrů ze zásobníku.

Tato zdánlivá chyba je v pořádku neboť podprogram nebude volán příkazem USR, ale přímo programem obsluhy přerušení, který si zakrátko napíšeme v BASICu. Přerušení nevyžaduje PLA, protože nepředává strojovému podprogramu žádné parametry a zásobník tudíž zůstává v pořádku.

Projděme si podprogramem detailně. Nejprve naplníme akumulátor

hexadecimálním číslem $42, které v barvovém systému ATARI specifikuje tmavě červenou barvu. Toto číslo vzniká součtem 16ti násobku barvy a jasu. Číslo $42 tedy znamená barvu 4 a jas 2. Toto číslo uložíme do hardwarového registru pro barvu pozadí v GRAPHICS 0, který je na adrese $D018 a má jméno COLPF2. Je důležité pochopit, proč používáme právě tento hardwarový registr a ne stínový barvový registr na adrese 710.

Pokud uložíme číslo (jako např. $42), které reprezentuje barvu, do stínového barvového registru v buňce 710, barva stínítka se změní na červenou a zůstane červená až do té doby, kdy opět změníme obsah buňky 710. To však není to, co jsme chtěli docílit v popisovaném podprogramu. Chceme, aby se změnila barva pouze dolní poloviny obrazovky, zatímco horní část má zůstat modrá. Musíme vědět, že hardwarový registr $D018 je naplňován ze svého stínového registru 710 padesátkrát za sekundu. Během každého vertikálního zatemňovacího impulsu přečte ATARI hodnotu v registru 710 a uloží tuto hodnotu do hardwarového registru $D018. To znamená, že padesátkrát za sekundu se nastaví modrá barva stínítka, protože číslo, uschované v registru 710 (je to 148), říká počítači, že požadujeme modrou barvu. Co tedy dělá náš podprogram?

Padesátkrát za sekundu, mezi kreslením snímků na obrazovku, se ATARI říká, že barva pozadí je modrá. Náš podprogram říká témuž hardwarovému registru, že po určitém počtu řádků má být barva pozadí červená a tedy zbývající linky jsou napsány červeně. Co se teď stane na konci snímku, když začne nový zatemňovací impuls: ATARI vezme hodnotu 148 a uloží ji do hardwarového registru a změní opět barvu pozadí na modrou, poté náš podprogram změní spodní část na červenou atd. Výsledek toho všeho je, že horní část obrázku je modrá a spodní červená. Pokud bychom místo $D018 v řádku 220 použili adresu 710, celá obrazovka by byla trvale červená.

Mezi naplněním akumulátoru požadovanou hodnotou barvy a jejím uložením do hardwarového registru je řádek 210, zapisující na adresu $D40A. Ta je pro používání přerušení z display-listu velmi důležitá.

Představme si elektronový paprsek, který přebíhá obrazovku zleva doprava. Pokaždé, když dorazí na pravý okraj, skočí zpět na levý okraj a začne kreslit další linku. Jestliže děláme něco jako změnu barvy pozadí, chceme mít jistotu, že ke změně dojde na začátku řádku a ne někde uprostřed. Pokud bychom jednoduše vložili novou barvu do hardwarového registru, změna barvy paprsku by nastala v tom místě, kde právě byl paprsek v okamžiku změny hodnoty v buňce $D018. Abychom tomu zabránili, uložíme v řádku 210 libovolnou hodnotu do buňky WSYNC. Na hodnotě, kterou ukládáme, přitom vůbec nezáleží, výslednou akci způsobí samotný akt ukládání. Ať je do buňky WSYNC ukládána jakákoliv hodnota, počítač jednoduše počká na horizontální sychronizační impuls, než bude pokračovat. Tato horizontální synchronizace nastane v okamžiku, kdy je elektronový paprsek mimo obrazovku a čeká na začátek další linky. Po synchronizaci provede počítač řádek 220, který uloží požadovanou hodnotu do hardwarového registru. Touto metodou se docílí, že ke změně barvy dojde vždy na začátku linky a ne někde uprostřed.

Zbytek programu jednoduše obnoví původní obsah akumulátoru a vrátí se do místa, kde byl program v okamžiku přerušení pomocí instrukce RTI v řádku 270.

Instalace obslužného programu přerušení vyžaduje určité znalosti programování v BASICu, protože sám o sobě nemůže změnu barvy udělat. Prohlédněme si program v BASICu pro implementaci obslužného programu:

```
10 GOSUB 20000:REM Zavedení do paměti
20 HIBYTE=INT(ADR(SIMPDLI$)/256):REM Kde je DLI podprogram
30 LOBYTE=ADR(SIMPLI$)-256*HIBYTE:REM Dolní byte adresy
```

```
40 POKE 512,LOBYTE:REM Nový dolní byte vektoru
50 POKE 513,HIBYTE:REM Nový horní byte
60 DL=PEEK(560)+256*PEEK(561):REM Kde je display-list
70 POKE DL+12,PEEK(DL+12)+128:REM Dosaď bit 7 přerušení
80 POKE 54286,192:REM Povol přerušení
90 END:REM Změna barvy zůstává
20000 DIM SIMPLI$(13):REM Přemístitelný program v řetězci
20010 FOR I=1 TO 13:REM Délka podprogramu
20020 READ A:REM Vezmi byte
20030 SIMPDLI$(I,I)=CHR$(A):REM Ulož jej do řetězce
20040 NEXT I:RETURN:REM Konec ukládání
20050 DATA 72,169,66,141,10,212,141,24,208,104
20060 DATA 64,246,243
```

Jak můžete vidět, nejprve uložíme napsaný podprogram do řetězce (v řádcích 20000 až 20060). Dále musíme zjistit, kam BASIC tento řetězec uložil a rozdělit adresu na dolní a horní byte. Pak řekneme počítači, kde je podprogram uložen, takže když narazí na přerušení z display-listu, bude vědět, kde najít program, který musí provést. Tato informace je v ATARI uložena vždy na adresách 512 a 513. Proto jsme v řádcích 40 a 50 uložili vypočtenou adresu do buněk 512 a 513.

V řádku 60 najdeme display-list. Řádek 70 nastaví bit 7, který signalizuje přerušení z display-listu, ve 12. bytu display-listu. Stejně snadno bychom mohli změnit barvu níže na stínítku, kdybychom místo DL+12 použili třeba DL+20. Nesnažte se jej změnit v některém ze dvou bytů, které ukazují na videopaměť (DL+4 nebo DL+5) nebo v některém ze 2 bytů, které ukazují na začátek display-listu (poslední 2 byty).

Řádek 80 je velice důležitý! I když jsme až doposud udělali vše, co je třeba pro přerušení z display-listu, ještě jsme ATARI neoznámili, že jej chceme povolit. To děláme až v řádku 80. Tato instrukce je nutná před tím než přerušení začne pracovat a máte-li potíže s uvedením obslužného programu do provozu, otestujete právě tento řádek před hledáním chyby ve vlastním strojovém podprogramu.

Komplikovanější příklad.

DLI podprogram, řízený tabulkou

Je řada možností, k nimž můžeme přerušení z display-listu (DLI) použít. Některé z nich jsou:

1. Změna barvy pozadí
2. Změna barvy znaků
3. Změna celého souboru znaků (uložením stránkové adresy do hardwarového registru - $D409, ne 756!)
4. Inverze znakového souboru - může být užitečná při kreslení hracích karet: nakreslí se polovina, invertuje se znakový soubor a nakreslí se spodní polovina (hardwarový registr $D401)
5. Simulace pohybu horizontu posouváním DLI.

Je možné využívat řadu dalších možností, omezením je pouze vaše představivost. Uvedeme zde ještě jeden příklad, jednoduše proto, abychom ukázali, jak implementovat trochu komplikovanější podprogram pro přerušení z display-listu. Jen si uvědomte, že času je málo a udržujte program kompaktní a krátký, jak je jen možné.

Tento příklad nám také ukáže použití tabulky. Dosadíme přerušení z display-listu do každého řádku displeje GRAPHICS 0 a změníme barvu pozadí za každým vyrobeným řádkem. Abychom toho docílili, musíme hodnoty barev postupně číst z tabulky a ve vhodnou dobu ukládat do

hardwarového registru barvy pozadí. Při příštím průchodu musíme vyzvednout další hodnotu z tabulky pro další řádek displeje. Naši tabulku vytvoříme ve stránce 4, ale mohla by být i ve stránce 6 nebo kdekoliv jinde v chráněné části paměti, jak už jsme si řekli. Podprogram v Assembleru následuje:

```
0100 ; ******************************
0110 ; nastav pocatecni podminky
0120 ; ******************************
0130  *= $600
0140 COLPF2 =$D018
0150 WSYNC = $D40A
0160 OFFSET = $0400
0170 ; ******************************
0180 ; uschovani registru
0190 ; ******************************
0200  PHA ;uchovej akumulator
0210  TYA ;  a registr Y
0220  PHA
0230 ; ******************************
0240 ; vlastni podprogram
0250 ; ******************************
0260  LDY OFFSET ;vezmi poc.  ofset
0270  LDA OFFSET+2,Y ;vezmi barvu
0280  STA WSYNC ;cekej na hor.synch.
0290  STA COLPF2 ; zmen barvu
0300  INC OFFSET ;pro dalsi barvu
0310  LDA OFFSET ;skoncili jsme?
0320  CMP OFFSET+1 ;zde je poc.barev
0330  BCC SKIP ; ne skonci pprog.
0340  LDA #0 ;ano znovu prvni barva
0350  STA OFFSET ;oprav ofset
0360 ; ******************************
0370 ; nezapomen obnovit registry!
0380 ; ******************************
0390 SKIP PLA ;priprav obnoveni Y
0400  TAY ; obnov registr Y
0410  PLA ;obnov akumulator
0420  RTI ;konec obsluhy preruseni
```

Všimněte si rozdílu mezi tímto programem a předchozím programem pro obsluhu přerušení z display-listu. Protože program používá jak akumulátor tak registr Y, musíme oba uchovat v zásobníku. To uděláme pomocí PHA a pak přenesením Y do akumulátoru pomocí TYA a dalšího PHA.

Hlavní rozdíl mezi oběma obslužnými programy je v řádcích 260 až 270 a 300 až 350. Nejprve naplníme registr Y z OFFSET v řádku 260. Tato hodnota představuje index do tabulky barev, která začíná na adrese $402 a pokračuje od ní nahoru v paměti. Pokud je OFFSET roven 5, vezmeme z tabulky šestou barvu (pamatujte: první barva má nulu). Tu uložíme do hardwarového registru barvy pozadí. V řádcích 300 až 350 zvýšíme hodnotu OFFSET a zkontrolujeme, zda jsme vyčerpali všechny barvy. Pokud ne, skončíme, pokud ano, před ukončením ještě nastavíme OFFSET na nulu. Všimněme si, že v buňce $401 (OFFSET+1) je počet barev v tabulce, takže podle něj můžeme testovat ukončení.

Protože jsme uschovali původní hodnoty registru X a akumulátoru, musíme je obnovit. V řádcích 390 až 420 to uděláme v obráceném pořadí jejich uchování.

Uvedeme nyní program v BASICu, kterým můžeme náš obslužný program

vyvolat.

```
10 GOSUB 20000:REM nacti prog.do stringu
20 HI=INT(ADR(TABLEDLI$)/256):LO=ADR(TABLEDLI$)-HI*256:
   REM adresa programu obsl. preruseni
30 GRAPHICS 0:SETCOLOR 1,0,0:REM zaciname s cernym pozadim
40 RESTORE 270:REM abychom cetli spravna data
50 FOR I=0 TO 27:REM pocet dat v tabulce
60 READ A:REM vezmi byte
70 POKE 1026+I,A:REM uloz barvu do tabulky ve strance 4
80 NEXT I
90 POKE 1024,0:REM zacinej s ofsetem 0
100 POKE 1025,27:REM zapis pocet barev
140 DL=PEEK(560)+256*PEEK(561)+6:REM normalni instrukce
    zacinaji v 7 bytu display-listu
150 DLBEG=DL-6:REM zacatek display/listu
160 FOR I=0 TO 2:REM prvni tri byty jsou instrukce
pro 8 volnych linek
170 POKE DLBEG+I,240:REM dosad bit pro DLI
    i do preskakovanych radku
180 NEXT I:REM konec cyklu
190 POKE DLBEG+I,194:REM dosad DLI take pro instrukci "load
    memory scan"
200 FOR I=DL TO DL+22:REM zmen vsechny dvojky na 130
210 POKE I,130:REM nastav bit pro preruseni do vsech
    instrukci
220 NEXT I:REM konec cyklu
230 POKE 512,LO:POKE 513,HI:REM rekni ATARI, kde je obsluzny
    program
240 POKE 54286,192:REM povol preruseni
250 LIST :REM napis neco, co bychom si mohli prohlizet
260 END :REM konec
20000 DIM TABLEDLI$(35):REM deklaruj retezec
20010 RESTORE 20060:REM abychom cetli spravna data
20020 FOR I=1 TO 35:REM pocet bytu v obsluznem programu
20030 READ A:REM vezmi byte
20040 TABLEDLI$(I,I)=CHR$(A):REM uloz byte na misto
      v retezci
20050 NEXT I:RETURN :REM konec retezce
20060 DATA 72,152,72,172,0,4,185,2,4,141
20070 DATA 10,212,141,24,208,238,0,4,173,0
20080 DATA 4,205,1,4,144,5,169,0,141,0
20090 DATA 4,104,168,104,64
```

Podprogram na řádku 20000 uloží náš obslužný program do řetězce. Pak zjistíme, kde je řetězec uložen a rozložíme jeho adresu na dolní a horní byte. Příkaz GRAPHICS 0 zajistí, že se vytvoří display-list tak, jak jej potřebujeme. Počáteční barvu pozadí nastavíme černou. Pak uložíme požadované hodnoty barev pozadí do tabulky ve stránce 4 byte po bytu. Změnou dat v řádku 270 můžeme získat jiný barevný vzor. Pohrajte si s těmito hodnotami, uvidíte, že je snadné vytvářet ve vašich programech zajímavé efekty. Do buňky $400 (dekadicky 1024) uložíme nulu, protože chceme, aby náš podprogram začínal od první barvy v tabulce. Pokud bychom tam uložili jiné číslo, řekněme 10, celé barevné spektrum by se posunulo nahoru po obrazovce a začínali bychom jedenáctou barvou a končili desátou.

Pak vyhledáme začátek display-listu a začátek příkazů pro GRAPHICS 0 (2 jako instrukce display-listu) a dosadíme nejvyšší bit každé instrukce v display-listu, čímž nastavíme přerušení z

display-listu v každém řádku. Poznamenejme, že můžeme dokonce dosadit bit přerušení i v prvních třech instrukcích display-listu, které každá pouze vynechávají 8 volných linek. Použitím tohoto programu dáváme každé skupině 8 linek jinou barvu. V řádku 230 říkáme počítači, kde je náš obslužný program přerušení a v dalším řádku toto přerušení povolíme. Příkaz LIST v řádku 250 jenom vypíše na obrazovku nějaký text na kterém můžeme pozorovat efekt, vytvořený obslužným programem.

Ještě poznámka o přerušení z display-listu: počítače ATARI používají WSYNC k vytvoření zvuku, který doprovází každé stisknutí klávesy na klávesnici. Programy, které hodně využívají přerušení z display-listu jako např. tento, mohou být rušeny mačkáním kláves. Nejjednodušší řešení je nevyžadovat ve svém programu vstup z klávesnice. Můžete např. vybírat pomocí menu ovladačem nebo používat tlačítka START, SELECT nebo OPTION pro výběr možností. Dále musíme ještě poznamenat, že SYSTEM RESET ukončí jakákoliv přerušení z display-listu, protože vytvoří pro GRAPHIC 0 nový display-list.

Přerušení při vertikálním zatemňovacím impulsu (VBI).

Druhý běžně používaný typ přerušení, použitý v ATARI, je přerušení od vertikálního zatemňovacího impulsu, o němž jsme již mluvili. S jeho využitím je možné na počítači ATARI používat multiprocessing, při němž se současně provádějí dva programy. I když s použitím přerušení od vertikálního impulsu nemůžeme vytvořit skutečný multiprocessing, je možné spustit dva programy tak, že jeden je zpracováván v normálním čase a druhý během vertikálního přerušení. Zvenčí to bude vypadat jako by se prováděly současně.

Pěkným příkladem použití tohoto způsobu je program EASTERN FRONT. Program "přemýšlí" o svých tazích během vertikálního přerušení a zpracovává tahy hráče v normálním čase. Čím déle přemýšlí hráč o svém tahu, tím více má počítač času na upřesnění svých tahů.

Další běžný příklad na použití vertikálního přerušení pro multiprocessing je v řadě programů pro ATARI. Jistě jste si všimli hudby, která se hraje v pozadí. Tato hudba vůbec nezpomaluje hru, protože je hrána pouze během vertikálního přerušení. Náš další program ukáže, jak se to dělá, i když na velmi jednoduchém příkladě. Ačkoliv je program přemístitelný, uložíme jej do stránky 6. Teď již víte, jak konvertovat program do řetězce a můžete to udělat snadno sami, chcete-li.

Pro vertikální přerušení jsou zapotřebí dvě části. Jednak je to samozřejmě vlastní obslužný program. Druhou je krátký program pro instalaci obslužného programu.

Normálně přejde ATARI při každém vertikálním zatemňovacím impulsu do programu, který se provádí při každé takové příležitosti. tento program je složen ze dvou částí. První se říká okamžitý a druhé zpožděný obslužný program vertikálního přerušení. Vektor okamžitého programu je na adrese $0222. Je to 2 bytová adresa, na kterou počítač skočí k provedení každého okamžitého obslužného programu a normálně ukazuje na obslužný program na adrese $E45F. Tento program končí skokem podle vektoru na adresách $0224 a $0225, které obsahují adresu zpožděného obslužného programu, a normálně ukazuje na $E462. Můžeme si to znázornit následovně:

VBI-> $0222-> (SYSVBV)$E45F-> $0224-> (XITVBV)$E462-> RTI

Obslužné programy vertikálního přerušení vykonávají řadu

důležitých funkcí jako obsluhu systémových hodin, kopírování stínových registrů, čtení ovladače a mnohé další. Proto je v dalším textu uváděn způsob, jak obslužný program rozšíříme o náš vlastní podprogram. Musíme si však uvědomit, že máme k dispozici jen velmi málo času a že musíme šetřit každou instrukcí.

Pokaždé když měníme nějaký vektor, setkáváme se s určitým potenciálním nebezpečím. U vektoru vertikálního přerušení, které se spouští 50krát za sekundu a může nastat v libovolném okamžiku vzhledem k provádění našeho programu, je uvedené potenciální nebezpečí zvláště vysoké. Nejlépe je popíšeme na jednoduchém příkladu. Víme, že byte, uložený na $0222 je $5F a byte na $0223 je $E4. Předpokládejme, že chceme tuto hodnotu změnit na $0620 namísto $E45F; nejprve změníme buňku $0222 na $20 a pak změníme $0223 na 6. To vypadá jednoduše. Předpokládejme ale, že po změně v buňce $0222 na $20 a ještě než stačíme změnit obsah $0223 dojde k vertikálnímu přerušení. Provede se skok na adresu v těchto dvou buňkách, která je ale nyní $E420, protože jsme změnili obsah jedné ale ne druhé. Počítač odejde do neznáma, protože na $E420 není nic rozumného. Abychom tento problém obešli, ATARI dodává vlastní program pro změnu vektoru vertikálního přerušení a tím zamezí vzniku uvedeného problému. Abychom viděli, jak pracuje, uvedeme si potřebné instrukce:

```
LDY #$20        ;dolní byte
LDX #$06        ;horní byte
LDA #07         ;pro zpožděný vektor
JSR SETVBV      ;dosaď vektor
RTS             ;vše hotovo
```

Pokud bychom chtěli změnit vektor okamžitého obslužného programu, naplnili bychom před skokem do programu SETVBV ($E45C) akumulátor hodnotou 6. To je vše. Uvědomte si, že váš obslužný program musí být na místě před provedením instalačního programu. Pokud tam není, program havaruje během padesátiny sekundy po provedení instrukce .

Délka obslužného programu je samozřejmě omezena. Zpožděný program může být maximálně dlouhý okolo 20000 strojových cyklů a okamžitý program nemůže být delší než asi 2000 cyklů. Je-li váš program delší, počítač havaruje, protože displej a počítač nebudou schopny nadále udržet synchronizaci.

Když jsme se nyní rozhodli použít zpožděný obslužný program a víme, jak jej nainstalovat, podívejme se na program, který nám zahraje v době vertikálního přerušení nějakou hudbu a proberme si ho podrobněji.

```
                0100 ; ********************
                0110 ; definice symbolických jmen
                0120 ; ********************
0000            0130        *=      $0600
00C0            0140 COUNT1 =       $00C0
0224            0150 VVBLKD =       $0224
00C2            0160 COUNT2 =       $00C2
E45C            0170 SETVBV =       $E45C
0660            0180 MUSIC  =       $0660
E462            0190 RETURN =       $E462
D200            0200 SND    =       $D200
D201            0210 VOL    =       $D201
                0220 ; ********************
                0230 ; PLA pro uvolnění zásobníku
                0240 ; ********************
0600 68         0250        PLA
                0260 ; ********************
```

```
                0270 ; nulování čitačů
                0280 ; ********************
0601 A900       0290        LDA   #0
0603 85C0       0300        STA   COUNT1     ;časovací čitač pro noty
0605 85C2       0310        STA   COUNT2     ;která nota se hraje
                0320 ; ********************
                0330 ; instalace vektoru
                0340 ; ********************
0607 A020       0350        LDY   #$20       ;dolní byte adresy
                                              programu
0609 A206       0360        LDX   #$06       ;horní byte adresy
060B A907       0370        LDA   #07        ;požadujeme zpožděný
                                              vektor
060D 205CE4     0380        JSR   SETVBV     ;instaluj vektor
0610 60         0390        RTS              ;instalace hotova
                0400 ; ********************
                0410 ; vlastní program VBI
                0420 ; ********************
0611            0430        *=    $0620
0620 E6C0       0440        INC   COUNT1     ;časování noty
0622 A6C0       0450        LDX   COUNT1     ;je nota skončena?
0624 E00C       0460        CPX   #12        ;jestliže >=12,
                                              je skončena
0626 9005       0470        BCC   NO         ;ještě není u konce
0628  A900      0480                   LDA   #0         ;ano - nastav
hlasitost=0
062A 8D01D2     0490        STA   VOL        ;zvuk vypojen
062D E00F       0500 NO     CPX   #15        ;uběhlo 15/50 sekundy?
062F B003       0510        BCS   PLAY       ;ano, hraj další notu
0631 4C62E4     0520        JMP   RETURN     ;ne, ať pokračuje
0634 A900       0530 PLAY   LDA   #0         ;nuluj čitač
0636 85C0       0540        STA   COUNT1     ;délky trvání noty
0638 A6C2       0550        LDX   COUNT2     ;pořadí noty
063A BD6006     0560        LDA   MUSIC,X    ;v tabulce
063D 8D00D2     0570        STA   SND        ;dosaď kmitočet
0640 A9A6       0580        LDA   #$A6       ;tvar=10 ($A)
0642 8D01D2     0590        STA   VOL        ;hlasitost=6
0645 E6C2       0600        INC   COUNT2     ;připrav další notu
0647 A6C2       0610        LDX   COUNT2     ;vyčerpali jsme všechny?
0649 E008       0620        CPX   #08        ;jestliže=8,
                                              skončili jsme
064B 9004       0630        BCC   DONE       ;ještě ne
064D A900       0640        LDA   #0         ;ano, nastav čitač not
064F 85C2       0650        STA   COUNT2     ;      znovu na začátek
0651 4C62E4     0660 DONE   JMP   RETURN     ;vše hotovo
                0670 ; ********************
                0680 ; tabulka not
                0690 ; ********************
0654            0700        *=    $0660
0660 F3         0710        .     BYTE  243, 243, 217, 243, 204,
                                        243, 217, 243
0661 F3
0662 D9
0663 F3
0664 CC
0665 F3
0666 D9
0667 F3
```

Inicializační podprogram vynuluje 2 čítače, jeden jako index noty, která se má hrát a druhý pro určení její délky. Pak instaluje vektor, ukazující na váš obslužný program, namísto normálního zpožděného obslužného programu. Vlastní obslužný program začíná na $0620 (řádek 430). Nejprve zvětšíme čítač trvání noty. Dosáhne-li 12, ukončíme notu, jinak nota pokračuje. Nota může být vypnuta uložením nuly do hardwarového registru, který řídí hlasitost jako v řádku 490.

Aby mezi notami zůstala krátká mezera, počkáme než čítač dosáhne 15. Pro novou notu uložíme 0 do čítače délky noty a vezmeme pořadové číslo noty, která se má hrát jako další. Toto číslo použijeme jako index do tabulky not na adrese $0660 (řádek 710). Jestliže COUNT2 obsahuje 0, bude se hrát třetí nota v pořadí. V řádku 560 se vyzvedne nota z tabulky a další tři řádky ji zahrají. Pak zvýšíme COUNT2 pro další notu a v řádcích 610 až 630 zjistíme, zda je melodie u konce; pokud ano, začneme s melodií znovu vynulováním čítače COUNT2.

Podprogram opustíme skokem na RETURN na adrese $E642, tím ukončíme obslužný program normálním zpožděným obslužným programem. Pokud bychom pro náš program použili okamžitý obslužný program, skočili bychom na konci na $E45F nebo, pro opravdu dlouhý program, na $E462, čímž bychom vyloučili celé normální zpracování vertikálního přerušení, ale získali spoustu času pro náš vlastní obslužný program.

Instalace v BASICu je docela jednoduchá, jak teď uvidíme:

```
10 GOSUB 19000:REM Načti inicializační podprogram
20 GOSUB 20000:REM Načti program VBI
30 GOSUB 21000:REM Načti tabulku not
40 X=USR(1536):REM Zapoj hudbu
50 END:REM Hudba se nevypne
19000 RESTORE 19050:REM Abychom četli správná data
19010 FOR I=1536 TO 1552:REM Inicializační podprogram
19020 READ A:REM Načti byte
19030 POKE I,A:REM Ulož jej
19040 NEXT I:RETURN:REM Načteno
19050 DATA 104,169,0,133,192,133,194,160,32,162
19060 DATA 6,169,7,32,92,228,96
20000 RESTORE 20050:REM Pro jistotu
20010 FOR I=1568 TO 1619:REM Podprogram VBI
20020 READ A:POKE I,A:REM Ulož na místo
20040 NEXT I:RETURN:REM Načteno
20050 DATA 230,192,166,192,224,12,144,5,169,0
20060 DATA 141,1,210,224,15,176,3,76,98,228
20070 DATA 169,0,133,192,166,194,189,96,6,141
20080 DATA 0,210,169,166,141,1,210,230,194,166
20090 DATA 194,224,8,144,4,169,0,133,194,76,98,228
21000 RESTORE 21050:REM Pro jistotu
21010 FOR I=1632 TO 1639:REM Délka tabulky
21020 READ A:POKE I,A:REM Ulož na místo
21040 NEXT I:RETURN:REM Načteno
21050 DATA 243,243,217,243,204,243,217,243
```

Tento program po načtení programů a tabulky pomocí USR inicializuje program a přitom instaluje příslušný vektor. Obslužné programy i tabulka jsou načteny do stránky 6 a hudba se aktivuje na řádku 40. A je to! Máte hudbu, která vás povzbudí při vysilujícím programování. Hudba bude hrát až do nejbližšího SYSTEM RESET nebo do doby, kdy změníte vektor obslužného programu na původní hodnotu.

Hudba hraná tímto programem je ve skutečnosti značně omezená. Všechny noty jsou téže délky; např samé čtvrťové nebo samé půlové. A dále, používá se pouze jeden hlas. Na ATARI jsou k dispozici i mnohem

komplikovanější možnosti, které vám dovolí použít složitou vícehlasou hudbu. A teď můžete takový program dokonce vytvořit sami.

Poslední poznámka k přerušení od vertikálního zatemňovacího impulsu: obzvlášť účelné využití této možnosti je čtení ovladače a pohyb hráčů po stínítku. Uložením takového programu do vertikálního přerušení můžeme odstranit jednu z částí programu v BASICu, které spotřebovávají nejvíce času a zároveň umožníme číst polohu ovladače a změnit polohu hráče 50 krát za sekundu, aniž bychom akci v reálném čase zpomalili. Jako cvičení můžete přepsat program pro čtení ovladače z Kapitoly 7 na program, volaný ve vertikálním přerušení.

Jemné posouvání (scrolling)

Musíme si ještě něco říci o posledních dvou bitech v instrukcích display-listu: bitech, které dovolují jemný horizontální a vertikální posuv (bity 4 a 5). Jemné posouvání dovoluje programátorovi vytvářet nejzajímavější a vzrušující efekty na ATARI: programy, které zdánlivě nekonečně posouvají obraz za stínítkem. Jeden z nejhezčích příkladů jemného posouvání je možno vidět v programu EASTERN FRONT, o kterém již byla řeč. Detailní mapa východní Evropy, jejíž velikost přesahuje několik normálně velkých obrazovek, se může libovolně posouvat, všude na celé mapě se něco děje.

Uvedeme si nyní příklad na jemný horizontální posuv a promluvíme si o jemném vertikálním posuvu tak, abyste si mohli psát vlastní podprogramy pro jemné posouvání. Horizontální posuv má jeden problém, o němž musíme pohovořit nejdříve. Jak už víte, normální display-list v modu GRAPHICS 0 obsahuje kod 2 pro každý řádek, čímž se říká ANTIC, že dalších 40 bytů videopaměti se má interpretovat jako text, který má být příslušně umístěn na stínítku. K potížím dojde, když chceme infromaci posunout doleva. Prohlédněme si problém graficky:

```
     Číslo sloupce
                    111111...2222223333333333
       01234567890123456...4567890123456789
.
.
.
     ř5 aaaaaaaaaaaaaaaa...aaaaaaaaaaaaaaaa
     ř6 bbbbbbbbbbbbbbbb...bbbbbbbbbbbbbbbb
     ř7 cccccccccccccccc...cccccccccccccccc
.
.
.
```

Pokud zobrazujeme pouze 40 znaků, není to problém. Jak ale můžeme posunou obrazové okénko za tuto informaci? Jestliže se např. pokusíme posunout okénko doprava (informaci doleva), jaký bude poslední znak na každém řádku? Řádek 5 bude teď končit b, řádek 6 c atd. To není opravdové horizontální posouvání, ale směs horizontálního a vertikálního.

Abychom docílili opravdového horizontálního posouvání, potřebujeme speciální formu display-listu. Musíme vytvořit display-list, který bude mít místo na více než 40 znaků na řádku, tak, abychom při posuvu mohli vidět informaci, která byla předtím schována mimo stínítko. Naštěstí už víme, jak takový display-list vytvořit. Budeme pro každý řádek muset specifikovat vlastní ukazatel do videopaměti a pro každý řádek musíme rezervovat mnohem více než 40 bytů. Navrhneme display-list, který bude mít každý řádek dlouhý 250 znaků, takže naše videopaměť bude více než 6krát širší než normální

stínítko GRAPHICS 0. To nám dá na posouvání spoustu místa. Obrazovka bude vypadat následovně:

```
     Číslo sloupce na obrazovce
                    11...33333333
          012345678901...23456789
.
.
     aaaaaaaaaaaaaaaa...aaaaaaaaaaaaaaaa
     bbbbbbbbbbbbbbbb...bbbbbbbbbbbbbbbb
     cccccccccccccccc...cccccccccccccccc
.
.
```

Ze znázornění vidíme, že velikost paměti pro každý řádek je větší než vlastní stínítko. To nám dává místo pro posuv stínítka se strany na stranu přes data, aniž by se nám přesouvala a mezi b atd.

Další vlastností našeho display-listu je, že každá instrukce LMS musí mít dosazený bit 4, takže musíme přičíst 16 k instrukci LMS 64. A samozřejmě musíme přičíst instrukci ANTICu pro interpretaci dat. V tomto případě použijeme GRAPHICS 0 (modus ANTIC 2), takže musíme přičíst 2. Celkový součet je 64+16+2, neboli 82, což bude instrukce pro každý řádek našeho vlastního display-listu.

Mohli bychom vytvořit náš display-list z BASICu jako výše, ale pustme se do experimentu a napišme program v Assembleru, který pro nás display-list vytvoří. Nový display-list uložíme na stránku 6, kde bude v bezpečí. Připomeňme si, že každý diplay-list začíná 24 prázdnými linkami, pokračuje 24 řádky kodu ANTIC než instrukce JVB ukončí program. Program, který takový display-list vytvoří, vypadadá následovně:

```
                    0100 ; ********************
                    0110 ; zacatek a symb.nazvy
                    0120 ; ********************
0600                0130         *=    $600
0600                0140 DLIST   =     $0600
0058                0150 SAVMSC  =     $58        ;
0230                0160 SDLSTL  =     $230       ;adresa DL
E45C                0170 SETVBV  =     $E45C      ;dosazeni VB vektoru
                    0180 ; ********************
                    0190 ; inicializacni podprogram
                    0200 ; pro vytvoreni display-listu
                    0210 ; a vlozeni posouvaciho
                    0220 ; podprogramu do vertikalniho
                    0230 ; preruseni
                    0240 ; ********************
0600 68             0250 INIT    PLA              ;uvolni stack
0601 A970           0260         LDA  #$70        ;8 volnych linek
0603 8D0006         0270         STA  DLIST       ;  do prvnich 3
0606 8D0106         0280         STA  DLIST+1     ;  radku
0609 8D0206         0290         STA  DLIST+2     ;  display-listu
060C A018           0300         LDY  #24         ;pocet radku v DL
060E A203           0310         LDX  #3          ;dosad citac
0610 A952           0320         LDA  #82         ; LMS+GR.0+SCROLL
0612 9D0006         0330         STA  DLIST,X     ;do DL
0615 E8             0340         INX              ;zvetsi citac
0616 A558           0350         LDA  SAVMSC      ;adr.videopameti
0618 9D0006         0360         STA  DLIST,X     ;do display-listu
061B E8             0370         INX              ;zvetsi citac
061C A559           0380         LDA  SAVMSC+1    ;horni byte
```

```
061E 38      0390         SEC              ;priprava odecitani
061F E918    0400         SBC  #24         ;udelej misto pro display
0621 9D0006  0410         STA  DLIST,X     ;uloz do DL
0624 E8      0420         INX              ;zvetsi citac
0625 88      0430         DEY              ;jeden radek ukoncen
0626 A952    0440 LOOP    LDA  #82         ;LMS+hor.posuv
0628 9D0006  0450         STA  DLIST,X     ;do DL
062B E8      0460         INX              ;zvetsi citac
062C BDFD05  0470         LDA  DLIST-3,X   ;posledni adresa
062F 18      0480         CLC              ;priprav scitani
0630 69FA    0490         ADC  #250        ;radek je 250 bytu
0632 9D0006  0500         STA  DLIST,X
0635 E8      0510         INX              ;zvetsi citac
0636 BDFD05  0520         LDA  DLIST-3,X   ;horni byte
0639 6900    0530         ADC  #0          ;viz vyklad
063B 9D0006  0540         STA  DLIST,X     ;do DL
063E E8      0550         INX              ;zvetsi citac
063F 88      0560         DEY              ;dalsi radek u konce
0640 D0E4    0570         BNE  LOOP        ;konec?NE
0642 A941    0580         LDA  #65         ;ANO, instrukce JVB
0644 9D0006  0590         STA  DLIST,X     ;do DL
0647 E8      0600         INX              ;zvetsi citac
0648 A900    0610         LDA  #0          ;str.6 dolni byte
064A 9D0006  0620         STA  DLIST,X     ;do DL
064D 8D3002  0630         STA  SDLSTL      ;oznam take ATARI
0650 E8      0640         INX              ;zvetsi citac
0651 A906    0650         LDA  #6          ;str.6 horni byte
0653 9D0006  0660         STA  DLIST,X     ;do DL
0656 8D3102  0670         STA  SDLSTL+1    ;oznam ATARI
             0680 ; ********************
             0690 ; vloz posouvaci podprogram
             0700 ; do zpozd.podpr. vert. preruseni
             0710 ; ********************
0659 68      0720         PLA              ;adresa podprogramu
065A AA      0730         TAX              ;do registru X
065B 68      0740         PLA              ;konec adresy
065C A8      0750         TAY              ;do registru Y
065D A907    0760         LDA  #$07        ;zpozdeny vektor
065F 205CE4  0770         JSR  SETVBV      ;dosad vektor
0662 60      0780         RTS              ;konec
```

Začneme zařazením tří instrukcí, z nichž každá znamená vynechání 8 prázdných linek ($70) na začátek nového display-listu. Dále vezmeme do registru Y 24 pro počítání, kolik řádků display-listu jsme již vytvořili. Do registru X dáme 3, protože chceme přeskočit první 3 instrukce $70. Další instrukce, kterou potřebujeme v display-listu, je 82, takže ji uložíme v řádcích 320 a 330. Pak zvětšíme čítač v registru X, protože jsme přidali byte do narůstajícího display-listu. S každým přidaným bytem zvětšíme registr X o 1.

Poněvadž každý řádek současně specifikuje LMS, jsou další 2 byty adresou videopaměti, odkud ATARI vezme informaci, kterou má zobrazit. Začátek display-listu má ukazovat na začátek videopaměti a tento ukazatel se vždy najde v buňkách $58 a $59, SAVMSC. Naše značně rozšířená videopaměť, více než 6krát větší než normální, potřebuje místo. Odečteme proto 24 stránek od horního bytu normální polohy videopaměti a tím získáme místo, které potřebujeme. Přenos informace z buňky $58 do nového display-listu je v řádcích 350 až 370 a přenos z $59 a zvětšení videopaměti je v řádcích 380 až 420. Poněvadž jsme ted ukončili řádek nového display-listu, který obsahuje instrukci LMS a

adresu, zmenšíme čítač řádek v registru Y (řádek 430).

Nyní vstoupíme do velkého cyklu od ř.440 do 570. Cyklus proběhne 23krát a pokaždé vytvoří další řádek display-listu. První vložená instrukce je 82, jak jsme řekli výše. Pak vezmeme dolní byte poslední adresy a přičteme k němu 250 (v řádcích 470 až 510). Připomeňme si použití CLC před každým sčítáním! To zvýší dolní byte druhého řádku videopaměti o 250 bytů, takže každý řádek bude 250 bytů namísto běžných 40.

Řádky 520 až 540 vypadají, jako by nedělaly nic, je to tak? Přičítají nulu k buňce v paměti a vrátí ji zpět. Mějme však na paměti, že v každé instrukci ADC se přičítá i přenos a my jsme bit přenosu od posledního přičítání nenulovali. Takže je-li výsledkem předcházejícího sčítání číslo > 255, dolní byte, uložený do display-listu v ř.500 bude ve skutečnosti roven součtu -256. V tom případě však bude dosazen bit přenosu do 1 a zvýší horní byte adresy o 1, když přičteme 0. Adresa bude pak ukazovat na správné místo v paměti. Je i jiná možnost, jak tuto operaci zapsat:

```
        LDA ADDR1
        CLC
        ADC #250
        STA ADDR1
        BCC PASS
        INC ADDR2
      PASS ...
```

V tomto případě pokud není bit přenosu v 1 po prvním sčítání, hodnota v ADDR2 se nezvětší; pokud je však první součet > 255, ADDR2 se zvýší o 1.

Smyčku ukončíme zmenšením čítače řádků v registru Y. Pokud jsme nedosáhli 0, máme ještě co dělat a smyčku zopakujeme. Pokud hodnota v Y dosáhla 0, skončili jsme s touto částí a potřebujeme ještě doplnit instrukci JVB aby ukazovala na začátek display-listu. To uděláme v řádcích 580 až 670. Všimněme si ještě řádků 530 a 670, ve kterých uložíme adresu nového display-listu do buněk $230 a $231, interních ukazatalů na display-list, které ATARI (a ANTIC) používá.

Aby posuv byl opravdu rychlý a jemný, umístíme náš podprogram do obslužného podprogramu vertikálního přerušení. Náš program v BASICu předá adresu posouvacího podprogramu inicializačnímu podprogramu a v řádcích 720 až 770 se tato adresa převezme ze zásobníku a zařadí se do zpožděného obslužného programu. Konečně se vrátíme do BASICu v řádku 780.

Nyní když jsme zkonstruovali display-list, musíme ještě napsat krátký podprogram ve strojovém jazyku, který uskuteční vlastní posuv. Pro lepší pochopení nejprve pohovoříme o mechanismu jemného posuvu. Znak v modu GRAPHICS 0 je 8 bitů široký. Hrubý posuv se uskutečňuje po znacích, v každém kroku jako by každý znak na stínítku poskočil o 1 místo vpravo nebo vlevo. My však požadujeme jemný posuv, při kterém je každý posuv o jeden bod či jeden bit v některém směru. U ATARI toho docílíme poměrně jednoduše s pomocí registru, který se jmenuje HSCROL ($D404). Odpovídající registr pro jemný vertikální posuv, který pracuje přesně stejně, se jmenuje VSCROL a je umístěn na $D405.

HSCROL může zajistit posuv bit po bitu pro 8 bitů, pak ale musí být vynulován. Je-li v HSCROL nula, poloha znaků je normální. Napíšeme-li do HSCROL 1, znak se posune o 1 bod doleva. Napsání 2 posune obraz o další bod atd., až do 7. V tomto okamžiku do HSCROL uložíme 0 a celý znak posuneme doleva o 1 celou posici změnou adresy v instrukci LMS pro každý řádek. Můžeme si to znázornit následovně:

```
Číslo zapsané do HSCROL
    0          1          2
.......I  .......I.  .....I..
.......I  .......I.  .....I..
.......I  .......I.  .....I..
.......I  .......I.  .....I..
.......I  .......I.  .....I..
.......I  .......I.  .....I..
.......I  .......I.  .....I..
.......I  .......I.  .....I..
```

Poté co jsme ukončili celý cyklus od 0 do 7 posunutím znaku o 1 celé místo, začneme nový od 0 do 7 atd. Budeme-li pokračovat, můžeme posunout celou šířku videopaměti. Ve skutečnosti podprogram, který si napíšeme, nebude testovat šířku videopaměti, takže bude pokračovat v posouvání třeba na konec paměti, necháme-li jej běžet dost dlouho. Můžete si prohlédnout operační systém svého ATARI novým a zcela odlišným způsobem!

Když teď víme, co chceme udělat, prohlédněme si program:

```
                0100 ; ******************
                0110 ; zacatek a symb.nazvy
                0120 ; ******************
0000            0130         *=    $600
0600            0140 DLIST   =     $600
D404            0150 HSCROL  =     $D404
E462            0160 XITVBV  =     $E462
                0170 ; ******************
                0180 ; uchovej akum. a X-reg.
                0190 ; ******************
0600 48         0200         PHA             ;uchovej akumulator
0601 8A         0210         TXA             ;prenes X-registr
0602 48         0220         PHA             ;a uchovej
                0230 ; ******************
                0240 ; nejprve jemne posouvani
                0250 ; ******************
0603 A207       0260         LDX  #7         ;7 bytu na znak
0605 8E04D4     0270 LOOP    STX  HSCROL     ;posuv prvniho
0608 CA         0280         DEX             ;priprav posuv dalsiho
0609 10FA       0290         BPL  LOOP       ;opakuj az do 8
060B A207       0300         LDX  #7         ;zmen registr posuvu
060D 8E04D4     0310         STX  HSCROL     ;  na zacatek
                0320 ; ******************
                0330 ; hruby posuv
                0340 ; ******************
0610 A200       0350         LDX  #0         ;citac
0612 BD0406     0360 LOOP2   LDA  DLIST+4,X  ;videopam.
0615 18         0370         CLC             ;priprav scitani
0616 6901       0380         ADC  #1         ;pricti 1
0618 9D0406     0390         STA  DLIST+4,X  ;v display-listu
061B BD0506     0400         LDA  DLIST+5,X  ;horni byte
061E 6900       0410         ADC  #0         ;pricti prenos
0620 9D0506     0420         STA  DLIST+5,X  ;v DL
0623 E8         0430         INX
0624 E8         0440         INX
0625 E8         0450         INX
0626 E048       0460         CPX  #72        ;24*3=72
0628 90E8       0470         BCC  LOOP2      ;Jeste neni konec
                0480 ; ******************
```

```
                0490 ; obnova registru
                0500 ; ****************
062A 68         0510        PLA             ;nejdrive X
062B AA         0520        TAX             ;  obnoven
062C 68         0530        PLA             ;pak akumulator
062D 4C62E4 0540        JMP  XITVBV     ;vystup z VB
```

Protože tento podprogram bude v obslužném podprogramu přerušení, musíme v řádcích 200 až 220 uschovat registr X a akumulátor. Dále, v řádcích 260 až 310 rychle proběhneme všemi 8 možnostmi HSCROL a před ukončením nastavíme čítač na 7. V řádcích 350 až 470 vstoupíme do další smyčky, která prochází celý display-list a zvyšuje každou adresu o 1 pro hrubý horizontální posuv. Pokud bychom dělali vertikální posuv, museli bychom přičítat ke každé adrese 250, abychom uskutečnili hrubý posuv nahoru o jeden řádek (nebo 40, pokud používáme display-list s normální šířkou řádku). V této smyčce používáme registr X jako čítač bytů spíše než řádků, takže v každém řádku jej musíme zvýšit o 3 (protože pro každý řádek v display-listu máme 3 byty).

Konečné v řádcích 510 až 530 obnovíme hodnoty registrů, uchované na začátku programu a v řádku 540 opustíme podprogram obsluhy vertikálního přerušení.

Můžeme i teď napsat velmi jednoduchý program v BASICu k použití obou podprogramů, které jsme si právě vyrobili:

```
10 GOSUB 20000:REM Vytvori retezec s modifikovanym DL
20 GOSUB 30000:REM Vytvori retezec s posouvacim podpr.
30 FOR I=34000 TO 40000 STEP 5:POKE I,86:NEXT I:REM Ulozi
   do pameti radky pro posuv
40 DUMMY=USR(ADR(DLSCROLL$),ADR(SCROLL$))
50 GOTO 50
20000 DIM DLSCROLL$(99):REM Retezec pro Display-list
20010 FOR I=1 TO 99:REM Delka retezce
20020 READ A:DLSCROLL$(I,I)=CHR$(A):REM Uloz do retezce
20040 NEXT I:RETURN :REM Zapsano
20050 DATA 104,169,112,141,0,6,141,1,6,141
20060 DATA 2,6,160,24,162,3,169,82,157,0
20070 DATA 6,232,165,88,157,0,6,232,165,89
20080 DATA 56,233,24,157,0,6,232,136,169,82
20090 DATA 157,0,6,232,189,253,5,24,105,250
20100 DATA 157,0,6,232,189,253,5,105,0,157
20110 DATA 0,6,232,136,208,228,169,65,157,0
20120 DATA 6,232,169,0,157,0,6,141,48,2
20130 DATA 232,169,6,157,0,6,141,49,2,104
20140 DATA 170,104,168,169,7,32,92,228,96
30000 DIM SCROLL$(48):REM Retezec pro VB podpr.
30010 FOR I=1 TO 48:REM Delka podpr.
30020 READ A:SCROLL$(I,I)=CHR$(A):REM ULOZ DO RETEZCE
30040 NEXT I:RETURN :REM Ulozeno
30050 DATA 72,138,72,162,7,142,4,212,202,16
30060 DATA 250,162,7,142,4,212,162,0,189,4
30070 DATA 6,24,105,1,157,4,6,189,5,6
30080 DATA 105,0,157,5,6,232,232,232,224,72
30090 DATA 144,232,104,170,104,76,98,228
```

Tento program nejdříve uloží do řetězců program pro vytvoření display-listu a posouvací podprogram pomocí podprogramů na řádcích 20000 a 30000. Řádek 30 pouze uloží do naší rozšířené videopaměti několik vertikálních čar, abychom měli co posouvat. Řádek 40 zřídí nový display-list použitím DLSCROLL$ a předá adresu SCROLL$ tomuto

podprogramu, takže může být zapojena do vertikálního přerušení. Protože nebudeme dělat nic jiného, než pozorovat posouvání, řádek 50 pouze udržuje program v běhu zatímco program ve vertikálním přerušení (náš posouvací podprogram) pokračuje ve své činnosti. Tím jsme skončili náš přehled o display-listu, videopaměti, obsluze přerušení a jemném posouvání. Teď byste měli být schopni psát dost složité podprogramy ve strojovém jazyce a snadno je využívat v programech v BASICu.

Kapitola 9:

Vstup a výstup na ATARI

V každém počítačovém systému jsou "vstup" a "výstup" termíny pro označení komunikace mezi mikroprocesorem a jakýmkoliv vnějším zařízením - klávesnicí, obrazovkou, tiskárnou, diskem, magnetofonem nebo podobným zařízením. Operační systém ATARI obsahuje podprogramy pro spolupráci s kterýmkoliv tímto zařízením na několika úrovních, tuto schopnost má řada mikroprocesorů. Co dělá systém ATARI jedinečným a z hlediska programátora tak snadno použitelným je, že se všemi vnějšími zařízeními se zachází jednotně a liší se jenom malými rozdíly ve vstupních a výstupních podprogramech.

Vstup je předávání informace z vnějšího světa, např. z klávesnice, do mikropočítače. Výstup je obrácený proces, kde informace postupuje z počítače ven, např. na tiskárnu. Ve zbytku této knihy budeme mluvit o ústředním systému vstupu a výstupu jako o CIO (Central Input-Output system).

Vektory v počítači ATARI

Již dříve jsme řekli, že techniky a programy používané v této knize budou pracovat na každém počítači ATARI, protože ATARI zaručuje neměnnost vektorů, odkazujících na podprogramy v operačním systému. Uvnitř operačního systému ATARI je tabulka skoků, obsahující adresy všech klíčových podprogramů, potřebných pro programování v jazyce assembler. Tato tabulka je na adresách $E450 až $E47F a v ATARI 130 XE obsahuje:

```
Adresa Instrukce
 E450  JMP $C6A3
 E453  JMP $C683
 E456  JMP $E4DF
 E459  JMP $C933
 E45C  JMP $C272
 E45F  JMP $C0E2
 E462  JMP $C28A
 E465  JMP $E95C
 E468  JMP $EC17
 E46B  JMP $C00C
 E46E  JMP $E4C1
 E471  JMP $F223
 E474  JMP $C290
 E477  JMP $C2C8
 E47A  JMP $FD8D
 E47D  JMP $FCF7
```

Je zřejmé, proč se jí říká tabulka skoků, protože je to tabulka adres, na které program skáče, chceme-li je vyvolat. Můžete se zeptat: "Proč neskákat přímo na dané adresy?". V odpovědi je klíč k psaní

programů, které pracují na všech typech počítačů ATARI (8 bitových). Předpokládejme, že místo skoku na $E456 bychom se rozhodli skákat přímo na $E4DF bez použití tabulky skoků. Vše by pracovalo hezky a náš program by fungoval. Ale, předpokládejme, že ATARI vyrobí nějaký nový počítač třeba 24800 XLTVB a operační systém se musí trochu upravit aby zvládl některé nové možnosti nového stroje. Náš program bude mít potíže. ATARI nikdy nezaručilo, že $E4DF tam zůstane navždy; jediné co zaručují je, že tabulka skoků bude vždy odkazovat na správnou adresu. Takže pokud bychom použili $E456 místo $E4DF, náš program bude pracovat vždy, protože je zajištěno, že $E456 se nezmění. Prohlédněme si jednotlivé vektory v tabulce skoků současně s jejich symbolickými jmény (která budeme používat pro tyto adresy ve všech našich programech).

```
Jméno Adresa Použití
DISKIV $E450 Inicializace obsluhy disku
DSKINV $E453 Obsluha disku
CIOV   $E456 Ústřední vstup a výstup
SIOV   $E459 Seriový vstup a výstup
SETVBV $E45C Dosazení obsluhy systemových časovačů
SYSVBV $E45F Zpracování přerušení od vertikálního
             zatemňovacího impulsu
XITVBV $E462 Výstup z vertikálního přerušení
SIOINV $E465 Inicializace seriového vstupu a výstupu
SENDEV $E468 Povolení výstupu na seriovou sběrnici
INTINV $E46B Zpracování přerušení
CIOINV $E46E Inicializace ústředního vstupu a výstupu
BLKBDV $E471 (Blackboard mode vector to memo pad mode)
WARMSV $E474 Teplý start (Po SYSTEM RESET)
COLDSV $E477 Studený start (Po zapnutí počítače)
RBLOKV $E47A Čtení bloku z kasety
CSOPIV $E47D Otevření kasety pro vstup
```

Některé z těchto vektorů použijeme v ukázkových programech a některé nepoužijeme vůbec; ale jejich znalost nám pomůže je využít vevlastních programech. Mnoho z nich využívá samotný operační systém.

K použití některého z těchto podprogramů v našem programu stačí použít JSR. Všechny jsou psány jako podprogramy a končí tedy instrukcí RTS, která vrátí řízení do našeho programu. Např. pro zavolání CIO, jednoduše napíšeme:

```
JSR  CIOV
```

Samozřejmě před tímto zavoláním je nutné připravit řadu věcí , o tom si hned řekneme; ale vlastní zavolání CIO nemůže být jednodušší.

Když jsme si řekli o vektorech v operačním systému, řekněme si ještě krátce o vektorech RAM a ROM. Jsou sepsány v následující tabulce se svými symbolickými jmény a krátkou poznámkou o jejich použití:

```
Jméno  Adresa Ukazuje na  Použití
CASINI $0002  různě       Inicializace samostatného programu,
                                 nataženého z kazety
DOSINI $000C  různě       Inicializace samostatného programu
                                 nataženého z disku
DOSVEC $000A  různě       Start programu nataženého z disku
VDSLST $0200  C0CE        Vektor přerušení z display-listu
VPRCED $0202  C0CD        nepoužito
VINTER $0204  C0CD        nepoužito
VBREAK $0206  C0CD        Vektor zpracování přerušení
```

```
                                    instrukcí BRK
VKEYBD $0208  FC19          -"-   klávesnicí
VSERIN $020A  EB2C          Vektor zpracování přerušení
                            signálu READY seriového vstupu
VSEROR $020C  EAAD          Vektor zpracování přerušení
                            signálu READY seriového výstupu
VSEROC $020E  EAEC          Vektor zpracování přerušení
                            ukončení seriového přenosu
VTIMR1 $0210  C0CD          Přerušení od syst. čítačů
VTIMR2 $0212  C0CD          -"-
VTIMR3 $0214  C0CD          -"-
VIMIRQ $0216  C030          .
VVBLKI $0222  C0E2          Vektor okamžité rutiny vertikálního
                            zatemňovacího běhu.
VVBLKD $0224  C28A           -"-     zpožděné      -"-
CDTMA1 $0226  .             .
CDTMA2 $0228  .             Adresa podprogramu pro
                            systémové časovače
BRKKY  $0236  C092          Vektor zpracování klávesy BREAK
RUNVEC $02E0  různě         Load and go startovací adresa
INIVEC $02E2  různě         Load and go inicializace
```

Většina z těchto adres odkazuje do operačního systému na obsluhu přerušení.

Na rozdíl od tabulky skoků nemohou být tyto adresy použity v instrukci JSR přímo. Protože však rovněž odkazují na podprogramy operačního systému, chtěli bychom je používat pomocí JSR. Správný způsob jak to udělat, je použít JSR na instrukci, která použije JMP nepřímo na příslušnou adresu. Např. chceme použít skok do podprogramu podle adresy v DOSINI. Správné použití ilustruje příklad:

```
40  JSR  MYSPOT
45  .
50  .
60 MYSPOT JMP (DOSINI)
```

Po provedení JSR na MYSPOT, RTS v příslušném podprogramu vrátí řízení na řádek 45 v našem programu.

Když jsme nyní viděli, jak psát univerzální programy pro počítače ATARI, řekněme si něco o filosofii CIO a naučme se psát programy pro interakci s vnějším světem.

Řídicí blok vstupu/výstupu (IOCB)

CIO systém v ATARI používá dva řídicí bloky. Jsou to řídicí blok vstupu a výstupu IOCB a řídicí tabulka zařízení (handler table). Řekněme si něco o každém zvlášť a pak uvidíme, jak spolupracují a vytvářejí funkční CIO.

IOCB je část paměti ve stránce 3, která obsahuje informaci, dodanou programátorem, kterou informujeme ATARI o zařízení, které chceme použít a jaká informace se bude předávat. Každý IOCB zabírá 16 bytů a k dispozici je 8 IOCB. Jejich jména a adresy jsou:

```
Jméno Umístění
IOCB0 $340 až $34F
IOCB1 $350 až $35F
```

```
IOCB2 $360 až $36F
IOCB3 $370 až $37F
IOCB4 $380 až $38F
IOCB5 $390 až $39F
IOCB6 $3A0 až $3AF
IOCB7 $3B0 až $3BF
```

Některé z těchto IOCB jsou používány normálně systémem, i když jako programátoři je můžeme použít standardním způsobem i změnit je pro vlastní potřebu. Normálně jsou použity OSem pouze 3 a většinou není třeba je předefinovávat, protože si můžeme vybrat z dalších 5. Tři použité OS jsou následující:

1. IOCB0, editor z obrazovky. Posláním výstupu na IOCB0 posíláme informaci obrazovkovému editoru. Tento IOCB zároveň řídí textové okénko v kterémkoliv grafickém modu s dělenou obrazovkou.
2. IOCB6, obrazovkový displej pro grafické mody vyšší než 0. Tento IOCB se používá pro všechny grafické příkazy jako PLOT, DRAWTO, FILL a další.
3. IOCB7, použitý k podpoře příkazu LPRINT v BASICu, který předává informaci na tiskárnu po vydání tohoto příkazu. V praxi většina výstupu z BASICu na tiskárnu používá některý z ostatních IOCB, protože LPRINT se příliš často nepoužívá; jestliže se pro tiskárnu otevře vlastní IOCB, je k dispozici více možností formátování.

Jak jste si již asi všimli, BASIC používá čísla IOCB (0,6 a 7) pro směrování informace na tato zařízení jako např. při psaní v GRAPHICS 1 nebo 2 příkazem:

```
PRINT #6;"HELLO"
```

16 bytů IOCB a jejich poloha vzhledem k začátku IOCB je v následující tabulce:

Jméno	Poloha	Délka	Popis
ICHID	0	1	Index to tabulky jmen zařízení pro tento IOCB
ICDNO	1	1	Číslo zařízení
ICCOM	2	1	Příkazový byte, určuje akci, která se má provádět
ICSTA	3	1	Status zařízení po provedení akce
ICBAL/H	4,5	2	2bytová adresa vyrovnávací paměti
ICPTL/H	6,7	2	Adresa-1 podprogramu pro PUT znaku
ICBLL/H	8,9	2	Délka vyrovnávací paměti
ICAX1	10	1	První doplňkový byte
ICAX2	11	1	Druhý doplňkový byte
ICAX3/4	12,13	2	Doplňkové byty 3 a 4 - pro NOTE a POINT v BASICu
ICAX5	14	1	Pátý doplňkový byte rovněž pro NOTE a POINT
ICAX6	15	1	Zatím nepoužitý doplňkový byte

Jednoduchý příklad

Než půjdeme do detailů o všech možných bytech, potřebných pro jednotlivé funkce IOCB, bude vhodné si udělat jednoduchý příklad, který nám pomůže pochopit jejich použití. Vezměme si jednoduchý příklad v BASICu a převeďme jej do ekvivalentního programu v Assembleru. Chceme naprogramovat ekvivalent příkazu v BASICu:

```
CLOSE #4:OPEN #4,6,0,"D:*.*"
```

Pro začátek potřebujeme vědět, že řídicí byte, uložený v ICCOM musí být $C pro příkaz CLOSE a 3 pro OPEN a kromě toho otevření seznamu disku vyžaduje 6 v bytu ICAX1. Jak vypadá program pro otevření takového souboru:

```
                0100 ; ********************
                0110 ; symbolicke nazvy
                0120 ; ********************
0000            0130        *=    $600
0341            0140 ICDNO  =     $0341
0342            0150 ICCOM  =     $0342
0344            0160 ICBAL  =     $0344
0345            0170 ICBAH  =     $0345
034A            0180 ICAX1  =     $034A
E456            0190 CIOV   =     $E456
                0200 ; ********************
                0210 ; close #4 pro jistotu
                0220 ; ********************
0600 A240       0230        LDX   #$40        ;pro IOCB #4
0602 A90C       0240        LDA   #$C         ;ridici byte pro CLOSE
0604 9D4203     0250        STA   ICCOM,X     ;X->IOCB #4
0607 2056E4     0260        JSR   CIOV
                0270 ; ********************
                0280 ; ted otevreme directory
                0290 ; ********************
060A A240       0300        LDX   #$40        ;opet #$40 = IOCB #4
060C A901       0310        LDA   #1          ;cislo disku
060E 9D4103     0320        STA   ICDNO,X     ;sem uloz cislo disku
0611 A903       0330        LDA   #3          ;ridici byte pro OPEN
0613 9D4203     0340        STA   ICCOM,X     ;do IOCB
0616 A906       0350        LDA   #6          ;directory disku
0618 9D4A03     0360        STA   ICAX1,X     ;uloz zde
061B A929       0370        LDA   #FILE&255   ;viz vyklad
061D 9D4403     0380        STA   ICBAL,X     ;dolni byte buf.
0620 A906       0390        LDA   #FILE/256   ;viz vyklad
0622 9D4503     0400        STA   ICBAH,X     ;horni byte adresy
0625 2056E4     0410        JSR   CIOV        ;udelej OPEN
0628 60         0420        RTS               ;vse hotovo
                0430 ; ********************
                0440 ; jeste jmeno souboru
                0450 ; ********************
0629 44         0460 FILE   .BYTE "D:*.*",$9B
062A 3A
062B 2A
062C 2E
062D 2A
062E 9B
```

Jak pro OPEN tak pro CLOSE naplníme do registru X hodnotu $40 pro adresování IOCB4.(Pokud bychom chtěli použít IOCB3, dali bychom do registru X $30 a podobně pro další IOCB). Pak uložíme řídicí byte $C do ICCOM v tomto IOCB a skok do podprogramu JSR na adresu CIOV provede uzavření souboru. Je vždy účelné před OPEN udělat CLOSE pro případ, že IOCB byl otevřen pro nějaký jiný účel. Pokud by soubor již byl otevřen, po návratu z CIO by se ohlásila chyba. Chybu můžete testovat

po každém volání operačního systému a případně odskočit do vlastního programu ošetření chyby tak, že po JSR na CIOV otestujete bit N stavového registru. Proto bychom měli za JSR CIOV zařadit instrukci BMI ERROR, ale pro účely této diskuse budeme předpokládat, že vše dopadlo dobře. Ve svých vlastních programech však takovéto předpoklady neděleite.

Pro otevření souboru (OPEN) uložíme 1 do ICDNO v IOCB4 pro disk č.1 a příkazový byte 3 do ICCOM v IOCB4 a ještě 6 do ICAX1. Před zavoláním CIO, musíme ještě nastavit adresu na jméno souboru, který chceme otevřít. Toto jméno je na řádku 460 s návěštím FILE. Znak $9B za jménem je hexadecimální kod pro návrat (RETURN), a má vždy ukončovat jméno souboru nebo zařízení jako např. S: nebo P:.

Pro nastavení adresy jména souboru ji musíme rozdělit na dolní a horní byte. Dolní byte je adresa AND 255, zapsáno #FILE&255. Logický součin s 255 nám dodá pouze dolní byte adresy. Horní byte adresy získáme dělením adresy 256, jak je uděláno v řádku 390. Dolní a horní byte jsou uloženy do ICBAL a ICBAH resp. a zavolání CIO v řádku 410 provede vlastní otevření souboru.

Tento jednoduchý příklad nejen demonstruje, jak otevřít seznam disku, ale také přesně ukazuje, jak je uděláno každé volání CIO ve vašem ATARI. Nejprve nastavíme příslušné byty v IOCB a pak jednoduše provedeme JSR na CIOV, ať chceme otevřít soubor, číst nějakou informaci z disku či pásky nebo poslat výstup na tiskárnu. Povšimněme si, že ne všech 16 bytů v IOCB se před voláním CIO musí nastavit. Pro některé příkazy stačí nastavit jeden nebo dva byty.

Podrobný popis jednotlivých bytů v IOCB.

Když jsme teď viděli, jak zapsat jednoduché volání ústředního systému pro vstup a výstup (CIO) v počítači ATARI, projdeme si veškerou informaci, která musí být v jednotlivých místech v IOCB pro vstupní a výstupní instrukce. Jednotlivé byty IOCB si projdeme v pořadí, v kterém jdou za sebou.

První byte ICHID slouží jako index do tabulky zařízení, takže z něj vždy můžete zjistit, které zařízení příslušný IOCB obsluhuje. Tento byte dosazuje OS a vy jej nebudete měnit. OS určí jeho hodnotu v průběhu příkazu OPEN a uloží sem příslušnou informaci.

ICDNO, číslo jednotky, se používá nejčastěji tehdy, když je v systému připojena více než jedna disková jednotka. Pro obsluhu každé jednotky je použit jiný IOCB a byte 2 v IOCB rozlišuje mezi použitými jednotkami. Je-li zde uložena 1, IOCB použije disk 1, podobně pro jednotky 2 až 4.

Řídicí byty pro různá zařízení, která mohou být připojena k ATARI, jsou následující:

Příkaz	Byte	Popis
Open	3	Otevři soubor
Get record	5	Načti řádek
Get character	7	Načti 1 nebo více znaků
Put record	9	Pošli řádek na výstup
Put character	11	Pošli na výstup 1 nebo více znaků
Close	12	Uzavři soubor
Status	13	Dodej stav zařízení
Draw line	17	Nakresli čáru v některém z grafických modů
Fill command	18	Vyplň část grafického stínítka barvou
Format disk	254	Formátuj disk

Čtvrtý byte IOCB je ICSTA, který je nastaven OSem při návratu z CIO. Stav je rovněž uložen do registru Y při návratu z jakéhokoliv volání CIO, takže váš program může podle obsahu registru Y nebo ICSTA určit úspěch či neúspěch každé vstupní nebo výstupní operace. Jakýkoliv záporný stav (hodnota větší než 128 nebo $80 hexadecimálně) indikuje, že při vstupní nebo výstupní operaci došlo k chybě.

Další dva byty IOCB slouží jako ukazatel na vyrovnávací paměť pro vstup nebo výstup a jsou v pořadí obvyklém pro 6502, dolní byte jako první. Jmenují se ICBAL a ICBAH. Vyrovnávací paměť (buffer) je část paměti, která obsahuje informaci, kterou chceme poslat na výstup nebo do které si přejeme umístit informaci ze vstupu. Chceme-li např. poslat text na tiskárnu, dosadíme ICBAL a ICBAH tak, aby ukazovaly na paměť, obsahující text, který chceme tisknout. Chceme-li přečíst do paměti diskový soubor, tyto byty IOCB dosadíme tak, aby ukazovaly na oblast v paměti, kam si přejeme umístit informaci z disku.

ICPTL a ICPTH slouží jako další 2-bytový ukazatel, ale v tomto případě odkazují na adresu podprogramu pro PUT minus 1. Každé zařízení, které může být otevřeno pro výstup, musí mít vlastní podprogram pro PUT, který říká počítači, jak tento přenos uskutečnit. Více si o tom povíme při popisu tabulky obslužného programu (handler table).

Další 2 byty IOCB jsou ICBLL a ICBLH, které obsahují délku vyrovnávací paměti v bytech. Jak uvidíme, ve zvláštním případě dosadíme délku rovnu 0 tím, že uložíme 0 jak do ICBLL tak ICBLH. V tomto zvláštním případě se informace nepřenáší do nebo z paměti, ale do nebo z akumulátoru.

Protože mnohá zařízení, která mohou být připojena k ATARI mají několik možných funkcí, musíme být schopni definovat v IOCB, kterou funkci použít. To uděláme pomocí bytu ICAX1, dalšího bytu IOCB. Následující tabulka vyjmenovává různé možné hodnoty ICAX1. V této tabulce TW označuje oddělené textové okénko na obrazovce, jaké vytváří příkaz v Basicu GRAPHICS 3, RE označuje povolení vstupu z obrazovky a RD znamená, že čtení z obrazovky není dovoleno.

Zařízení	Byte	Funkce
Obrazovka	8	Výstup na obrazovku
(Editor)	12	Vstup z klávesnice a výstup na obrazovku
	13	Vnucený vstup a výstup z obrazovky
Obrazovka	8	Výmaz obrazovky, ne TW, RD
	12	Výmaz obrazovky, ne TW, RE
	24	Výmaz obrazovky, TW, RD
	28	Výmaz obrazovky, TW, RE
	40	Bez výmazu, ne TW, RD
	44	Bez výmazu, ne TW, RE
	56	Bez výmazu, TW, RD
	60	Bez výmazu, TW RE
Klávesnice	4	Čtení -pozn. výst. neex.
Tiskárna	8	Zápis -pozn. vstup neex.
Magnetofon	4	Čtení
	8	Zápis
RS-232	5	Současné čtení (concurrent read)
	8	Blokový zápis
	9	Současný zápis (concurrent write)
	13	Současné čtení a zápis

```
                (conc. read and write)
Disk          4 Čtení
              6 Čtení seznamu disku
              8 Zápis nového souboru
              9 Zápis s připojením
                (append)
             12 Čtení a zápis v opravném
                modu (update)
```

Poslední byte IOCB, o kterém si zde řekneme, je ICAX2, který se používá pouze ve speciálních případech, jinak je roven 0. Při použití magnetofonu znamená hodnota 128 v ICAX2, že se použijí krátké mezibloková mezery na pásce, čímž se docílí rychlejšího čtení takto zapsané pásky. Hodnota 0 v ICAX2 znamená zápis normálních dlouhých meziblokových mezer.

Konečně, uložením příslušného čísla do ICAX2 při OPEN se specifikují grafické mody 0 až 11. V kombinaci s ICAX1, popsaným výše, dává ICAX2 programátoru v assembleru možnost plně řídit grafický modus, textové okénko, mazání obrazovky i čtení a zápis z obrazovky. Více se o tom dovíme v kapitole 10.

Tabulka obslužného programu (handler table).

Když jsme si nyní popsali jednotlivé části IOCB, popíšeme si stručně také tabulku obslužného programu, která společně s bloky IOCB tvoří vstupní a výstupní systém CIO. Pak se podíváme na několik příkladů, které nám ukáží, jak tuto informaci můžeme využít k provedení řady různých vstupních a výstupních příkazů z assembleru. Nejjednodušší způsob, jak prozkoumat tabulku obslužného programu je představit si ji jako krátký program v asssembleru asi takto:

```
0100 PRINTV = $E430
0110 CASETV = $E440
0120 EDITRV = $E400
0130 SCRENV = $E410
0140 KEYBDV = $E420
0150 ; ******************************
0160 ; začátek tabulky = $031A
0170 ; ******************************
0180  *= $031A
0190 .BYTE "P"      ;vektor
0200 .WORD PRINTV   ;  tiskárny
0210 .BYTE "C"      ;vektor
0220 .WORD CASETV   ;  magnetofonu
0230 .BYTE "E"      ;vektor
0240 .WORD EDITRV   ;  editoru
0250 .BYTE "S"      ;vektor
0260 .WORD SCRENV   ;  obrazovky
0270 .BYTE "K"      ;vektor
0280 .WORD KEYBDV   ;  klávesnice
0290 .BYTE 0        ;volné #1(DOS)
0300 .WORD 0,0
0310 .BYTE 0        ;volné #2(interface ATARI 850)
0320 .WORD 0,0
0330 .BYTE 0        ;volné #3
0340 .WORD 0,0
```

```
0350 .BYTE 0        ;volné #4
0360 .WORD 0,0
0370 .BYTE 0        ;volné #5
0380 .WORD 0,0
0390 .BYTE 0        ;volné #6
0400 .WORD 0,0
0410 .BYTE 0        ;volné #7
0420 .WORD 0,0
```

Každá položka v tabulce sestává z prvního písmene specifikovaného zařízení, následovaného vektorem, ukazujícím na místo v paměti, kde je uložena informace potřebná pro práci s tímto zařízením. Jak vidíte, je v tabulce ještě 7 volných položek, takže programátor může přidat jakékoliv zařízení, které pro nějaký účel potřebuje a bude se s ním zacházet stejně jako se zařízením již specifikovaným. O tabulce obslužného programu poznamenejme ještě jednu důležitou věc. Kdykoliv operační systém prohledává tabulku, prochází ji zdola nahoru! To je uděláno úmyslně a umožní vám to přidat např. vlastní obslužný program pro tiskárnu dolů do tabulky. Při prohledávání tabulky bude váš vektor nalezen jako první a také bude použit. Můžete tedy napsat vlastní obslužné programy pro tiskárnu a nahradit jimi normální jednoduše umístěním jiného P: do některé z dolních volných položek; za ně umístíme 2 bytový adresní vektor, ukazující na vaše nové obslužné programy.

Podívejme se ještě krátce na tabulku vstupních bodů programů (handler entry point table), na kterou ukazuje položka v tabulce obslužného programu. Např. vektor PRINTV, uvedený výše, ukazuje na další tabulku, tabulku vstupních bodů. Ve skutečnosti všechny výše uvedené vektory ukazují na své tabulky vstupních bodů a všechny tyto tabulky jsou uspořádány stejně. Obsahují adresy minus 1 podprogramů, pro následující funkce v uvedeném pořadí:

```
OPEN otevření zařízení
CLOSE uzavření zařízení
READ podprogram čtení
WRITE podprogram zápisu
STATUS zjištění stavu
SPECIAL speciální funkce, pokud jsou implementovány
```

Tato tabulka je vždy zakončena 3 bytovou instrukcí JMP na inicializační podprogram tohoto zařízení. Pamatujte si: adresy, které jsou v tabulce vstupních bodů neukazují na podprogramy pro OPEN a CLOSE, ale ukazují na adresu o 1 nižší než je začátek každého z těchto proprogramů. Je zjevně velmi důležité mít toto na paměti, když vytváříte svou vlastní tabulku vstupních bodů.

Jednoduchý program

Podívejme se, jak můžeme použít CIO pro jednoduchou funkci - psaní na obrazovku. Umíme to v BASICu, chceme-li psát na obrazovku řádek textu, musíme napsat příkaz např.:

```
PRINT "USPESNY ZAPIS!"
```

V assembleru je také docela jednoduché psát na obrazovku, když teď již známe použití IOCB a CIO. Jen si připomeneme - obrazovku nemusíme otevírat jako zařízení, pokud nechceme, protože IOCB0 je již

určen OS pro tento účel. Můžeme tedy do registru X vzít nulu a použít ji jako ofset na IOCB. Anebo můžeme přímo použít absolutní adresování, protože budeme používat první IOCB. V příkladu uvedeném dále použijeme registr X naplněný nulou hlavně z toho důvodu, abychom se seznámili s obvyklou procedurou umístění požadované informace do IOCB. Zde je program pro napsání řádku na obrazovku:

```
                0100 ; ********************************
                0110 ; symbolicka jmena
                0120 ; ********************************
0340            0130 ICHID  =    $0340
0341            0140 ICDNO  =    $0341
0342            0150 ICCOM  =    $0342
0343            0160 ICSTA  =    $0343
0344            0170 ICBAL  =    $0344
0345            0180 ICBAH  =    $0345
0346            0190 ICPTL  =    $0346
0347            0200 ICPTH  =    $0347
0348            0210 ICBLL  =    $0348
0349            0220 ICBLH  =    $0349
034A            0230 ICAX1  =    $034A
034B            0240 ICAX2  =    $034B
E456            0250 CIOV   =    $E456
0000            0260         *=  $600
                0270 ; ********************************
                0280 ; nyni nacteme pozadovana data
                0290 ; ********************************
0600 A200       0300         LDX  #0          ;pouzivame IOCB #0
0602 A909       0310         LDA  #9          ;pro PUT
0604 9D4203     0320         STA  ICCOM,X     ;ridici byte
0607 A91F       0330         LDA  #MSGa255    ;dolni byte MSG
0609 9D4403     0340         STA  ICBAL,X     ;  do ICBAL
060C A906       0350         LDA  #MSG/256    ;horni byte MSG
060E 9D4503     0360         STA  ICBAH,X     ;  do ICBAH
0611 A900       0370         LDA  #0          ;delka zpravy
0613 9D4903     0380         STA  ICBLH,X     ;horni byte
0616 A9FF       0390         LDA  #$FF        ; > delka zpravy
0618 9D4803     0400         STA  ICBLL,X     ; viz vyklad
                0410 ; ********************************
                0420 ; ted to napis na obrazovku
                0430 ; ********************************
061B 2056E4     0440         JSR  CIOV
061E 60         0450         RTS
                0460 ; ********************************
                0470 ; vlastni zprava
                0480 ; ********************************
061F 55         0490 MSG     .BYTE "USPESNY ZAPIS!",$9B
0620 53
0621 50
0622 45
0623 53
0624 4E
0625 59
0626 20
0627 5A
0628 41
0629 50
062A 49
062B 53
062C 21
```

062D 9B

Psaní na obrazovku je v BASICu tak snadné, že by nemělo smysl psát tento program jako podprogram pro BASIC, proto také neobsahuje obvyklou instrukci PLA. Protože budete potřebovat psát na stínítko při ladění programů v assembleru, tento program může být jedním z těch, které budete používat nejčastěji.

Abychom otestovali program po jeho zadání do počítače, jednoduše napíšeme ASM, abychom jej přeložili a je-li překlad hotov, napíšeme BUG, abychom vstoupili do modu DEBUG (ladění) Assembler-Editoru. Pak napíšeme G600, aby se program spustil od adresy $600. Byl-li program zadán správně, měla by se objevit věta USPESNY ZAPIS!, následovaná výpisem obsahu registrů 6502. Ten se vypisuje po vyvolání libovolného programu, který používá Assebler-Editor. Stejným způsobem budeme testovat každý z programů, uvedených v této knize. Vzniknou-li problémy, přezkoušejte, zda jste vše napsali správně.

POZOR !!! Vždy uložte své programy PREDTIM než se je pokusíte spustit!!! Protože pokud dopadnou špatně nebo pokud se zhroutí systém, nebudete muset znovu vyťukávat celý program.

V tomto programu jsme k výpisu celé zprávy na obrazovku použili příkazu výpis věty, protože jsme do ICCOM uložili 9. Adresa textu, který chceme zobrazit na obrazovce, je pak uložena do ICBAL a ICBAH jako dříve. Do horního bytu délky textu uložíme 0, ale do dolního bytu uložíme $FF.

Proč $FF, když celá zpráva je kratší než 20 bytů? Když se použije CIO v modu výpis věty, text je vypisován byte po bytu až se dorazí na konec vyrovnávací paměti nebo až se dojde ke znaku nový řádek ve vypisovaném textu. Všimněme si, že text, vytvořený v řádku 490, je ukončen bytem $9B, což je právě znak pro nový řádek. Tudíž na obrazovku se pošle text, návrat vozíku (nový řádek) a program se ukončí. Úmyslně jsme dosadili délku zprávy delší než skutečnou, protože jsme chtěli, aby $9B uvnitř vlastního textu ukončil výstup. Takto se nám nestane, že bychom neúmyslně text zkrátili dosazením ICBLL a ICBLH menším než jsme zamýšleli.

Je důležité si uvědomit, že jsme nedosazovali všechny byty v IOCB. Musíme vlastně dosadit pouze ty byty, které použije daný podprogram. Jak dále uvidíte, to platí u služebních programů obecně. Vlastní výstup na obrazovku je dosažen zavoláním podprogramu pro vstup a výstup v řádku 440 a RTS v dalším řádku vrátí řízení do Assembler-Editoru. Pokud by to byla část většího programu, pokračoval by od řádku 450 bez RTS.

Podívejme se na jiný způsob, jak napsat zprávu na obrazovku s použitím systému vstupu a výstupu. Namísto naplnění ICCOM hodnotou 9 pro zápis věty jej můžeme naplnit 11 pro zápis bytů. Ostatní byty IOCB se naplní jako minule s výjimkou ICBLL, do něhož dosadíme přesnou délku textu. Při počítání bytů textu nezapomeňte započítat byte pro $9B, návrat vozíku. Program pak bude vypadat následovně:

```
                0100 ; ******************************
                0110 ; symbolicka jmena
                0120 ; ******************************
0340            0130 ICHID  =    $0340
0341            0140 ICDNO  =    $0341
0342            0150 ICCOM  =    $0342
0343            0160 ICSTA  =    $0343
0344            0170 ICBAL  =    $0344
0345            0180 ICBAH  =    $0345
```

```
0345            0190 ICPTL  =    $0345
0347            0200 ICPTH  =    $0347
0348            0210 ICBLL  =    $0348
0349            0220 ICBLH  =    $0349
034A            0230 ICAX1  =    $034A
034B            0240 ICAX2  =    $034B
E456            0250 CIOV   =    $E456
0000            0260         *=  $600
                0270 ; ******************************
                0280 ; nyni nacteme pozadovana data
                0290 ; ******************************
0600 A200       0300         LDX  #0          ;pouzivame IOCB #0
0602 A90B       0310         LDA  #11         ;pro PUT byty
0604 9D4203     0320         STA  ICCOM,X     ;ridici byte
0607 A91F       0330         LDA  #MSG&255    ;dolni byte MSG
0609 9D4403     0340         STA  ICBAL,X     ;  do ICBAL
060C A906       0350         LDA  #MSG/256    ;horni byte MSG
060E 9D4503     0360         STA  ICBAH,X     ;  do ICBAH
0611 A900       0370         LDA  #0          ;delka zpravy
0613 9D4903     0380         STA  ICBLH,X     ;horni byte
0616 A90F       0390         LDA  #15         ;delka zpravy
0618 9D4803     0400         STA  ICBLL,X     ; dolni byte
                0410 ; ******************************
                0420 ; ted to napis na obrazovku
                0430 ; ******************************
061B 2056E4     0440         JSR  CIOV
061E 60         0450         RTS
                0460 ; ******************************
                0470 ; vlastni zprava
                0480 ; ******************************
061F 55         0490 MSG     .BYTE "USPESNY ZAPIS!",$9B
0620 53
0621 50
0622 45
0623 53
0624 4E
0625 59
0626 20
0627 5A
0628 41
0629 50
062A 49
062B 53
062C 21
062D 9B
```

Tento program vykoná totéž jako předcházející, ale jiným způsobem. Oba tyto programy napíší na stínítko zprávu, ukončenou návratem vozíku.

Existují případy psaní na stínítko, kdy nechceme, aby byl text ukončen návratem vozíku, jako např. čekáme-li na vstup nebo chceme-li na stínítko psát speciálním způsobem. V BASICu je k tomu příkaz PRINT, ukončený středníkem, který zabrání výstupu návratu vozíku. Jestliže např. chceme napsat na stínítko symbol > jako žádost o vstup, ale chceme, aby kursor zůstal na stejném řádku, v BASICu bychom to napsali takto:

```
PRINT ">";
```

V assembleru to bude vypadat následovně:

```
                0100 ; ******************************
                0110 ; symbolicka jmena
                0120 ; ******************************
0340            0130 ICHID  =    $0340
0341            0140 ICDNO  =    $0341
0342            0150 ICCOM  =    $0342
0343            0160 ICSTA  =    $0343
0344            0170 ICBAL  =    $0344
0345            0180 ICBAH  =    $0345
0346            0190 ICPTL  =    $0346
0347            0200 ICPTH  =    $0347
0348            0210 ICBLL  =    $0348
0349            0220 ICBLH  =    $0349
034A            0230 ICAX1  =    $034A
034B            0240 ICAX2  =    $034B
E456            0250 CIOV   =    $E456
0000            0260         *=  $600
                0270 ; ******************************
                0280 ; nyni nacteme pozadovana data
                0290 ; pro zapis 1 bytu
                0300 ; ******************************
0600 A90B       0310         LDX  #0        ;pro IOCB0
0602 A90B       0320         LDA  #11       ;pro PUT byty
0604 9D4203     0330         STA  ICCOM,X   ;ridici byte
0607 A900       0340         LDA  #0        ;delka zpravy
0609 9D4903     0350         STA  ICBLH,X   ;horni byte
060C A900       0360         LDA  #0        ;delka zpravy
060E 9D4803     0370         STA  ICBLL,X   ; viz vyklad
                0380 ; ******************************
                0390 ; ted to napis na obrazovku
                0400 ; ******************************
0611 A93E       0410         LDA  #62       ;pro ">"
0613 2056E4     0420         JSR  CIOV
0616 60         0430         RTS
```

Jestliže dosadíme délku vyrovnávací paměti nulovou (dosazením ICBLL i ICBLH na nulu), pak je na výstup poslán znak, obsažený v akumulátoru, aniž by byl následován návratem vozíku. To platí o všech zařízeních včetně disků, magnetofonů, tiskáren i o obrazovce a ukazuje nám velmi důležitou vlastnost počítačů ATARI: vstup a výstup jsou do značné míry nezávislé na zařízení. To znamená, že operační systém zachází se všemi zařízeními podobně, takže se nemusíme zvlášť učit, jak napsat text na obrazovku, potom jiný způsob, jak jej poslat na tiskárnu a další metodu, jak jej zapsat na disk. Metoda je stejná, jakmile byl IOCB otevřen pro příslušné zařízení. Abychom si to ukázali, podíváme se na program pro poslání stejné zprávy na tiskárnu.

Výstup na tiskárnu

Nejprve pro jistotu uzavřeme IOCB2, pak otevřeme s použitím IOCB2 tiskárnu a pošleme naši zprávu.

```
                0100 ; ******************************
                0110 ; symbolicka jmena
                0120 ; ******************************
0340            0130 ICHID  =    $0340
0341            0140 ICDNO  =    $0341
```

```
0342            0150 ICCOM  =    $0342
0343            0160 ICSTA  =    $0343
0344            0170 ICBAL  =    $0344
0345            0180 ICBAH  =    $0345
0346            0190 ICPTL  =    $0346
0347            0200 ICPTH  =    $0347
0348            0210 ICBLL  =    $0348
0349            0220 ICBLH  =    $0349
034A            0230 ICAX1  =    $034A
034B            0240 ICAX2  =    $034B
E456            0250 CIOV   =    $E456
0000            0260        *=   $600
                0270 ; ******************************
                0280 ; nejdrive zavreme a otevreme IOCB
                0290 ; ******************************
0600 A220       0300        LDX  #$20       ; pro IOCB2
0602 A90C       0310        LDA  #12        ;prikaz CLOSE
0604 9D4203     0320        STA  ICCOM,X    ;  do ICCOM
0607 2056E4     0330        JSR  CIOV       ;udelej CLOSE
060A A220       0340        LDX  #$20       ;opet IOCB2
060C A903       0350        LDA  #3         ;prikaz OPEN
060E 9D4203     0360        STA  ICCOM,X    ;  do ICCOM
0611 A908       0370        LDA  #8         ;vystup
0613 9D4A03     0380        STA  ICAX1,X    ;OPEN por vystup
0616 A94C       0390        LDA  #NAM&255   ;dolni byte zarizeni
0618 9D4403     0400        STA  ICBAL,X    ;ukazuje na "P:"
061B A906       0410        LDA  #NAM/256   ;horni byte
061D 9D4503     0420        STA  ICBAH,X
0620 A900       0430        LDA  #0
0622 9D4903     0440        STA  ICBLH,X    ;horni byte delky
0625 A9FF       0450        LDA  #$FF
0627 9D4803     0460        STA  ICBLL,X    ;> dolni byte delky
062A 2056E4     0470        JSR  CIOV       ;udelej OPEN
                0480 ; ******************************
                0490 ; nyni vytiskneme zpravu
                0500 ; ******************************
062D A220       0510        LDX  #$20       ;pouzivame IOCB2
062F A909       0520        LDA  #9         ;PUT vetu
0631 9D4203     0530        STA  ICCOM,X    ;  prikaz
0634 A94F       0540        LDA  #MSG&255   ;adresa MSG
0636 9D4403     0550        STA  ICBAL,X    ;  dolni byte
0639 A906       0560        LDA  #MSG/256   ;adresa MSG
063B 9D4503     0570        STA  ICBAH,X    ;  horni byte
063E A900       0580        LDA  #0         ;delka zpravy
0640 9D4903     0590        STA  ICBLH,X    ;  horni byte
0643 A9FF       0600        LDA  #$FF       ;> delka zpravy
0645 9D4803     0610        STA  ICBLL,X    dolni byte
0648 2056E4     0620        JSR  CIOV       ;vypis zpravu
064B 60         0630        RTS             ;konec
064C 50         0640 NAM    .BYTE "P:",$9B
064D 3A
064E 9B
064F 55         0650 MSG    .BYTE "USPESNY TISK!",$9B
0650 53
0651 50
0652 45
0653 53
0654 4E
0655 59
```

```
0656 20
0657 54
0658 49
0659 53
065A 4B
065B 21
065C 9B
```

Samozřejmě, pokud použijete pro výstup tiskárnu ATARI a chcete tisknout širokým písmem, musíte dosadit ICAX1 a ICAX2 před konečným zavoláním CIOV, ale to je triviální. Všimněte si, že jsme po vytisknutí zprávy nezavřeli IOCB2, takže chceme-li tisknout cokoli dalšího, jednoduše to pošleme pomocí IOCB2 bez nutnosti jej znovu otvírat. To ovšem samozřejmě znamená, že teď nemůžeme IOCB2 použít pro jiný účel např. pro disk. Chceme-li pracovat s diskem, můžeme použít některý jiný IOCB nebo nejdříve uzavřít IOCB2 a znovu otevřít pro naši diskovou operaci.

Poznamenejme ještě, že chceme-li, aby tiskárna vytiskla jeden znak bez dělicího návratu vozíku, můžeme použít speciální případ nulové délky, přesně tak, jako jsme to udělali pro výstup na obrazovku.

Výstup na disk

Abychom demonstrovali pružnost centrálního systému vstupu a výstupu na ATARI, neuvedeme ani program pro výstup stejného řádku do diskového souboru. Popíšeme metodu a sami budete schopni napsat příslušný program na první pokus a úplně sami! Jediná změna, kterou musíte udělat v programu uvedeném výše pro tiskárnu, je uvést jméno diskového souboru, který chcete otevřít. Program je tedy identický s předchozím s výjimkou řádku 640, který bude vypadat nějak takto:

```
640 NAM .BYTE "D:MYFILE.1",$9B
```

A to je vše. Teď vidíte krásu použití identických podprogramů CIO v Assembleru pro všechna zařízení podle vaší volby.

Vstup s použitím CIOV

Metoda, použitá v ATARI pro vstup je stejná jako pro výstup, ale zařízení musí být otevřeno pro vstup. Můžeme např. přečíst výše uvedenou zprávu ze souboru "D1:MYFILE.1" tak, že otevřeme soubor pro vstup použitím 3 v ICCOM a 4 v ICAX1 a nastavením ICBAL a ICBAH na adresu v paměti, na niž chceme zprávu přenést. Např. chceme-li, aby se zpráva uložila od adresy $680, dosadíme do ICBAL hodnotu #$80 a do ICBAH #6. Po zavolání CIOV budou buňky paměti počínaje $680 obsahovat byty zprávy, které pak mohou být zpracovány další částí programu.

Měli byste ocenit snadnost a jednoduchost používání vstupu a výstupu na počítači ATARI. Učit se každému zařízení zvlášť je běžné u řady jiných mikropočítačů; vstup a výstup mohou používat rozdílné podprogramy, každý se svými zvláštnostmi. Ústřední filozofie vstupu a výstupu, použitá v ATARI, pro nás značně zjednodušuje tento proces. Nyní tento systém můžete použít k výraznému rozšíření možností programování v assembleru.

Ještě poslední poznámka k vstupním a výstupním podprogramům: pokud otevřeme jeden IOCB pro vstup souboru z disku a druhý pro výstup na tiskárnu nebo na obrazovku, bude triviální přenášet informaci velmi

rychle z jednoho zařízení na druhé použitím jedné vyrovnávací paměti pro oba IOCB. Napsání výpisu z paměti nebo okopírování paměti do diskového souboru je jednoduché. Můžeme dokonce přenášet informaci z disku na obrazovku nebo na magnetofon.

Vstup a výstup bez použití CIO

Tři různé systémy pro vstup a výstup.

Kromě CIO existují ještě dva způsoby pro použití disku jako vstupní a výstupní jednotky, oba jsou součástí OS. Používají vektory DSKINV a SIOV na adresách $E453 a $E459 resp.

Tři metody pro vstup a výstup na disku si můžeme představit jako slupky cibule s více úrovněmi řízení. Vnější vrstva, která pro vás udělá většinu práce, je systém CIO; střední vrstva, která pro vás udělá část práce, je DSKINV; a vnitřní vrstva, v níž programátor musí udělat téměř vše, je systém SIO. Vektor pro seriový vstup a výstup SIOV se používá pro všechnu komunikaci, která se odehrává přes seriovou sběrnici (konektor s 13 kolíky na vašem počítači ATARI). Dokonce i systém CIO provádí vlastní vstupní a výstupní instrukce pomocí volání SIO. DSKINV, o němž si rovněž něco málo řekneme, také volá SIO pro vlastní vstup a výstup.

Typy diskových souborů

Disketa pro ATARI disk má v jednoduché hustotě 40 soustředných stop, trochu jako gramofonová deska. Na desce však stopy tvoří jednu spojitou spirálu, zatím co na disketě tvoří každou stopu samostatná kružnice. Každá stopa je rozdělena na 18 sektorů. Abychom si to znázornili, představme si disketu, rozkrájenou jako dort na 18 stejných dílků. Pak rozdělíme dort na 40 soustředných mezikruží. Každý kousek dortu je jeden sektor. Máme tedy 18x40, celkem 720 sektorů. V každém z těchto sektorů může ATARI uschovat 128 bytů informace.

Při formátování diskety se nevytvoří pouze těchto 720 sektorů, ale zároveň se vytvoří tabulka obsazení diskety (VTOC) a tabulka obsahu disku. Tabulku obsahu disku můžeme srovnat s obsahem knihy, v němž je uveden každý soubor (kapitola), zapsaný na disketě a číslo sektoru, kam je uložen (stránka). V tabulce VTOC se vede záznam o obsazených sektorech a o těch, které jsou volné, takže při ukládání souboru na částečně plný disk nepřepíšeme informaci, která je tam již obsažena. Když soubor zrušíme, jeho sektory jsou uvolněny ve VTOC, takže mohou být použity znovu. Poznamenejme ještě, že při zrušení souboru se doopravdy změní jediný - stavový byte. Prvních 5 bytů položky obsahu paměti jsou:

1. Stavový byte, který obsahuje stav souboru. Každý ze 4 bitů stavového bytu je použit pro uložení specifické informace o souboru:
   - Bit 0 je roven 1 v souboru, otevřeném pro výstup
   - Bit 5 je roven 1, je li soubor uzamčen
   - Bit 6 je roven 1 pro právě používaný soubor
   - Bit 7 je roven 1, byl-li soubor zrušen

2,3. Délka souboru v sektorech v obvyklém pořadí dolní/horní byte.
4,5. Číslo prvního sektoru na disketě v pořadí dolní/horní byte.

Máte-li k dispozici některý z mnoha pomocných programů pro práci s diskem (utility), můžete ještě obnovit zrušený program jednoduše změnou stavového bytu z $80 na $40. Jestliže jste však mezitím již na

disk něco uložili, tento postup nebude fungovat. Při zrušení souboru se změní i VTOC a sektory zrušeného souboru se uvolní pro další použití. A jestliže jste na disk mezitím psali, zjistíte, že některé ze sektorů dříve použitých v souboru, který chcete obnovit, již byly přepsány novou informací.

Pro účely této diskuse popíšeme dva různé typy souborů, použitých v počítačích ATARI. První a zdaleka nejčastější je zřetězený soubor, podobný jaký se vytvoří instrukcí v BASICu:

```
SAVE "D:GAME"
```

Nejprve se prohledá tabulka obsahu disku (directory). Protože na disku může být uloženo nejvýše 64 souborů, tento test zjistí, že je v tabulce ještě místo pro další soubor se jménem GAME. Jestliže se při testu tabulky zjistí, že na disku již soubor se jménem GAME existuje, je zrušen (pokud není uzamčen) a nový soubor nahradí starý. Předpokládejme, že je to první soubor se jménem GAME, který se má uložit a že v tabulce je místo. Pak se do prvního sektoru, o němž VTOC řekne, že je volný, zapíše prvních 125 bytů souboru GAME, i když do sektoru se vejde 128 bytů. Tím zbude místo na 3 byty, které doplní CIO a které dávají souboru jméno zřetězený soubor.

Tyto 3 byty obsahují následující informaci:

č. bytu:	125	126	127
	76543210	76543210	76543210
	č.soub.	odkaz dopř.	S poč.b.

Nejvyšších 6 bitů bytu 125 tvoří číslo souboru. Je-li např. soubor GAME čtvrtým souborem v seznamu, číslo v prvních 6 bitech bytu 125 je 3, protože číslování začíná od 0. Toto číslo se testuje při čtení každého sektoru pro kontrolu, zda sektor skutečně patří souboru GAME. Jestliže se při čtení souboru narazí na sektor s jiným číslem, ohlásí se chyba, která obvykle znamená, že došlo k porušení informace na disku a její oprava je velmi obtížná.

Dolní dva bity bytu 125 společně s 8 bity bytu 126 tvoří desetimístné binární číslo, obsahující číslo příštího sektoru souboru. Takže po přečtení prvního sektoru se odtud zjistí číslo dalšího, který se má přečíst atd., až je přečten poslední sektor. Proto se takovému souboru říká zřetězený soubor. Poslední sektor každého souboru obsahuje nulu jako číslo příštího sektoru, takže můžeme zjistit, že jsme už přečetli celý soubor.

Byte 127 každého sektoru obsahuje počet bytů, uložených v tomto sektoru. V každém sektoru, kromě posledního, je toto číslo rovno 125. Je-li v sektoru méně než 125 bytů, dosadí se roven 1 bit S, nejvyšší bit bytu 127, který znamená sektor kratší než 125 bytů.

Další hlavní typ souboru na disku se jmenuje sekvenční soubor a je ve své struktuře mnohem jednodušší. Nepoužívá se ani VTOC ani tabulka obsahu disku a při zápisu informace se použije všech 128 bytů v sektoru. Sektory takového souboru se čtou sekvenčně, sektor 3 se čte po sektoru 2 atd. První sektor takového souboru obsahuje zaváděcí adresu (kam se bude ukládat obsah souboru do paměti) a startovací adresu (kde se program po svém načtení nastartuje). Tento typ souboru se obvykle používá v komerčně dostupných hrách. Pokusíte-li se zobrazit seznam takového disku, dostanete nesmyslný výpis, protože na takovém disku ani tabulka obsahu nebyla vytvořena.

Pozn.překl.:

I zřetězené soubory, určené pro zavedení pomocí instrukce L systému DOS obsahují jednu nebo více zaváděcích adres a jednu nebo více startovacích adres. Na disketě, používané se sytémemem DOS 2.5,

jsou prakticky všechny soubory zřetězené , s výjimkou prvních několika málo prvních sektorů (obvykle tří) na disku, které tvoří sekvenční soubor a používají se pro natažení operačního systému DOS do paměti.

Použití různých systémů vstupu a výstupu.

Pro čtení zřetězeného souboru, jakým je např. program v BASICu, se obvykle použije CIO. Chceme-li však přečíst z disku konkrétní sektor, musíme použít DSKINV nebo SIO. Ukažme si, jak tyto úkoly splníme s použitím tří rozdílných způsobů volání vstupu a výstupu.

Nejprve otevřeme soubor na disku a načteme jej do paměti. Část programu, která otevře soubor, je velmi podobná programu, uvedenému výše pro čtení tabulky obsahu disku.

```
                0100 ; ****************************
                0110 ; symbolicka jmena
                0120 ; ****************************
0340            0130 ICHID  =    $0340
0341            0140 ICDNO  =    $0341
0342            0150 ICCOM  =    $0342
0343            0160 ICSTA  =    $0343
0344            0170 ICBAL  =    $0344
0345            0180 ICBAH  =    $0345
0346            0190 ICPTL  =    $0346
0347            0200 ICPTH  =    $0347
0348            0210 ICBLL  =    $0348
0349            0220 ICBLH  =    $0349
034A            0230 ICAX1  =    $034A
034B            0240 ICAX2  =    $034B
E456            0250 CIOV   =    $E456
0000            0260        *=   $600
                0270 ; ****************************
                0280 ; otevreme soubor OBJECT.COD
                0290 ; ****************************
0600 A220       0300        LDX  #$20       ; pro IOCB2
0602 A90C       0310        LDA  #12        ;prikaz CLOSE
0604 9D4203     0320        STA  ICCOM,X    ;  do ICCOM
0607 2056E4     0330        JSR  CIOV       ;udelej CLOSE
                0340 ; ****************************
060A A220       0350        LDX  #$20       ;opet IOCB2
060C A903       0360        LDA  #3         ;prikaz OPEN
060E 9D4203     0370        STA  ICCOM,X    ;  do ICCOM
0611 A904       0380        LDA  #4         ;vstup
0613 9D4A03     0390        STA  ICAX1,X    ;do ICAX1
0616 A900       0400        LDA  #0         ;0 do ICAX2 je pouze
0618 9D4B03     0410        STA  ICAX2,X    ;  pro jistotu
061B A94F       0420        LDA  #NAME              255 ;dolni byte jmena
061D 9D4403     0430        STA  ICBAL,X    ;adresa jmena
0620 A906       0440        LDA  #NAME/256 ;horni byte jmena
0622 9D4503     0450        STA  ICBAH,X    ;horni byte adresy
0625 2056E4     0460        JSR  CIOV       ;udelej OPEN
                0470 ; ****************************
0628 A220       0480        LDX  #$20       ;IOCB2
062A A900       0490        LDA  #0
062C 9D4403     0500        STA  ICBAL,X    ;dolni byte adresy
062F A950       0510        LDA  #$50       ;horni byte
```

```
0631 9D4503 0520         STA  ICBAH,X   ;adresa je $5000
0634 A9FF   0530         LDA  #$FF      ;vyrob delku vyr.pam.
0636 9D4803 0540         STA  ICBLL,X   ;  velmi dlouhou,
0639 9D4903 0550         STA  ICBLH,X   ;  aby se nacetl cely soubor
063C A905   0560         LDA  #5        ;cti vetu
063E 9D4203 0570         STA  ICCOM,X   ;ridici byte
0641 2056E4 0580         JSR  CIOV      ;cti cely soubor
            0590 ; ****************************
0644 A220   0600         LDX  #$20      ;IOCB2
0646 A90C   0610         LDA  #$C       ;pro CLOSE
0648 9D4203 0620         STA  ICCOM,X   ;ridici byte
064B 2056E4 0630         JSR  CIOV
064E 60     0640         RTS
            0650 ; ****************************
064F 44     0660 NAME    .BYTE "D1:OBJECT.COD",$9B
0650 31
0651 3A
0652 4F
0653 42
0654 4A
0655 45
0656 43
0657 54
0658 2E
0659 43
065A 4F
065B 44
065C 9B
```

Tento program načte celý program v jediné operaci. V řádcích 530 až 550 dosadíme délku vyrovnávací paměti rovnu 65535 neboli $FFFF. Podprogram CIO pak přečte celý soubor a zastaví buď po přečtení 65535 bytů (což není možné) nebo když narazí na byte návrat vozíku. Jestliže tedy náš soubor obsahuje nějaký byte $9B (návrat vozíku), čtení se ukončí a my nenačteme celý soubor. Jak se přes tento problém dostaneme?

Protože pravděpodobně nebudeme vědět, zda soubor obsahuje nějaké byty $9B, použijeme metodu, která je jistá a načte jakýkoliv soubor. Abychom toho docílili, budeme číst sektory po jednom (vždy 128 bytů) a budeme pokračovat, dokud nebude ohlášena chyba, k čemuž dojde na konci souboru. S použitím CIO se dozvíme, že došlo k chybě, protože při návratu z CIO bude dosazen bit N do 1. Prohlédněme si program, který provede tento typ čtení s použitím CIO:

```
            0100 ; ****************************
            0110 ; symbolicka jmena
            0120 ; ****************************
0340        0130 ICHID  =    $0340
0341        0140 ICDNO  =    $0341
0342        0150 ICCOM  =    $0342
0343        0160 ICSTA  =    $0343
0344        0170 ICBAL  =    $0344
0345        0180 ICBAH  =    $0345
0346        0190 ICPTL  =    $0346
0347        0200 ICPTH  =    $0347
0348        0210 ICBLL  =    $0348
0349        0220 ICBLH  =    $0349
034A        0230 ICAX1  =    $034A
034B        0240 ICAX2  =    $034B
```

```
E456            0250 CIOV   =    $E456
0000            0260        *=   $600
                0270 ; ********************************
                0280 ; otevreme soubor OBJECT.COD
                0290 ; ********************************
0600 A220       0300        LDX  #$20       ; pro IOCB2
0602 A90C       0310        LDA  #12        ;prikaz CLOSE
0604 9D4203     0320        STA  ICCOM,X    ;  do ICCOM
0607 2056E4     0330        JSR  CIOV       ;udelej CLOSE
                0340 ; ********************************
060A A220       0350        LDX  #$20       ;opet IOCB2
060C A903       0360        LDA  #3         ;prikaz OPEN
060E 9D4203     0370        STA  ICCOM,X    ;  do ICCOM
0611 A904       0380        LDA  #4         ;vstup
0613 9D4A03     0390        STA  ICAX1,X    ;do ICAX1
0616 A900       0400        LDA  #0         ;0 do ICAX2 je pouze
0618 9D4B03     0410        STA  ICAX2,X    ;  pro jistotu
061B A969       0420        LDA  #NAME                255 ;dolni byte jmena
061D 9D4403     0430        STA  ICBAL,X    ;adresa jmena
0620 A906       0440        LDA  #NAME/256  ;horni byte jmena
0622 9D4503     0450        STA  ICBAH,X    ;horni byte adresy
0625 2056E4     0460        JSR  CIOV       ;udelej OPEN
                0470 ; ********************************
0628 A220       0480        LDX  #$20       ;IOCB2
062A A900       0490        LDA  #0
062C 9D4903     0500        STA  ICBLH,X    ;horni byte delky
062F A980       0510        LDA  #$80       ;cti jednotlive segmenty
0631 9D4803     0520        STA  ICBLL,X    ;dolni byte delky
0634 A950       0530        LDA  #$50       ;horni byte adresy
0636 9D4503     0540        STA  ICBAH,X    ;  vyrov. pameti
0639 A905       0550        LDA  #5         ;cti vetu
063B 9D4203     0560        STA  ICCOM,X    ;ridici byte
063E A220       0570 LOOP   LDX  #$20
0640 A900       0580        LDA  #0         ;dolni byte vyr. pameti
0642 9D4403     0590        STA  ICBAL,X
0645 2056E4     0600        JSR  CIOV       ;cti prvni sektor
0648 3014       0610        BMI  FIN        ;na konci jdi na FIN
064A A220       0620        LDX  #$20       ;IOCB2
064C A980       0630        LDA  #$80       ;posun o 128 bytu
064E 9D4403     0640        STA  ICBAL,X    ;  ve vyr. pameti
0651 2056E4     0650        JSR  CIOV       ;cti dalsi sektor
0654 3008       0660        BMI  FIN        ;na konci jdi na FIN
0656 A220       0670        LDX  #$20       ;IOCB2
0658 FE4503     0680        INC  ICBAH,X    ;zvys adresu vyr.pam
065B 4C3E06     0690        JMP  LOOP       ;opakuj
                0700 ; ********************************
065E A220       0710 FIN    LDX  #$20       ;IOCB2
0660 A90C       0720        LDA  #12        ;pro CLOSE
0662 9D4203     0730        STA  ICCOM,X    ;ridici byte
0665 2056E4     0740        JSR  CIOV       ;udelej CLOSE
0668 60         0750        RTS             ;konec
                0760 ; ********************************
0669 44         0770 NAME   .BYTE "D1:OBJECT.COD",$9B
066A 31
066B 3A
066C 4F
066D 42
066E
066F
```

```
0670 43
0671 54
0672 2E
0673 43
0674 4F
0675 44
0676 9B
```

Pokračujeme ve čtení tak dlouho, dokud nedojde k chybě vstupu, v tom případě pak skočíme na FIN k uzavření souboru a ukončení programu. Musíte dávat pozor, aby nedošlo k jiné chybě než konec souboru, protože tento proram odskočí na FIN při každé chybě. Bylo by samozřejmě snadné napsat program, který nejprve otestuje číslo chyby v registru Y po návratu z volání CIOV a pak udělá příslušné ošetření. Poznamenejme, že v tomto programu musíme dělat některé další věci jako zvyšování adresy ve vyrovnávací paměti v řádcích 630,640 a 680, o to jsme se v prvním příkladu nemuseli starat. Kromě toho také musíme testovat ukončení čtení, což jsme v minulém programu nemuseli.

Pozn.překl.: Tak, jak je program napsán, nám nevyřeší problém ukončení čtení na $9B. Kromě toho se nepřesně uvádí, že čteme po sektorech. My už však víme, že sektor obsahuje jen 125 bytů informace. Navíc je při troše dobré vůle možné spočítat skutečný načtený počet bytů z hodnoty v buňkách ICBLL a ICBLH, kde je po skončení čtení rozdíl mezi zadanou délkou a počtem bytů skutečné načtených. Abychom překonali problém s $9B, použijeme na řádcích 560 resp. 550 instrukci LDA #7, čtení znaků.

Čtení s pomocí rezidentního obslužného programu disku.

Pro použití rezidentního obslužného programu musí programátor nastavit řídicí blok zařízení (DCB) způsobem, který je analogický nastavení IOCB pro použití s CIO. Symbolická jména pro DCB jsou následující:

```
0100 ; ******************************
0110 ; symbolicke nazvy pro SIO
0120 ; ******************************
0130 DDEVIC = $0300 ;ID pro ser.bus
0140 DUNIT  = $0301 ;cislo jednotky
0150 DCOMND = $0302 ;prikazovy byte
0160 DSTATS = $0303 ;stavovy byte
0170 DBUFLO = $0304 ;dolni byte adr.vyr.pameti
0180 DBUFHI = $0305 ;horni byte
0190 DTIMLO = $0306 ;timeout pro disk
0200 DBYTLO = $0308 ;dolni byte citace
0210 DBYTHI = $0309 ;horni byte
0220 DAUX1  = $030A ;doplnkovy #1
0230 DAUX2  = $030B ;doplnkovy #2
0240 SIOV   = $E459
0250 DSKINV = $E453
```

Třetí byte jak v IOCB tak v DCB je řídicí byte, i když vlastní obsah řídicího bytu se liší. Pátý a šestý byte obsahují adresy vyrovnávací paměti. Pro rezidentní program jsou povoleny tyto instrukce:

```
$21 formátování disku
$50 zápis sektoru
$52 čtení sektoru
$53 zjištění stavu
$57 zápis sektoru s ověřením
```

Je tedy zřejmé, že tento způsob práce s diskem je omezenější a tím i daleko jednodušší než s CIO. Podívejme se, jak můžeme použít DCB a rezidentní obslužný program disku přes DSKINV k přečtení informace z disku. Samozřejmě nebudeme číst normální soubory DOS, to jsou zřetězené soubory, a rezidentní program není pro jejich čtení vybaven. Předpokládejme tedy, že spíše než diskový soubor chceme číst sektory od $20 do $60 včetně. Program následuje:

```
                0100 ; ******************************
                0110 ; symbolicke nazvy pro SIO
                0120 ; ******************************
0300            0130 DDEVIC =    $0300      ;ID pro ser.bus
0301            0140 DUNIT  =    $0301      ;cislo jednotky
0302            0150 DCOMND =    $0302      ;prikazovy byte
0303            0160 DSTATS =    $0303      ;stavovy byte
0304            0170 DBUFLO =    $0304      ;dolni byte adr.vyr.pameti
0305            0180 DBUFHI =    $0305      ;horni byte
0306            0190 DTIMLO =    $0306      ;timeout pro disk
0308            0200 DBYTLO =    $0308      ;dolni byte citace
0309            0210 DBYTHI =    $0309      ;horni byte
030A            0220 DAUX1  =    $030A      ;doplnkovy #1
030B            0230 DAUX2  =    $030B      ;doplnkovy #2
E459            0240 SIOV   =    $E459
E453            0250 DSKINV =    $E453
0000            0260         *=  $600
                0270 ; ******************************
                0280 ; predpokladame, ze soubor zacina
                0290 ; sektorem $20 a konci $50
                0300 ; ******************************
0600 A900       0310         LDA #0
0602 8D0B03     0320         STA DAUX2      ;horni byte sektoru
0605 8D0803     0330         STA DBYTLO     ;doni byte delky
0608 A980       0340         LDA #$80       ;pro cteni
060A 8D0903     0350         STA DBYTHI     ;  sektoru po jednom
060D A950       0360         LDA #$50       ;horni byte
060F 8D0503     0370         STA DBUFHI     ;  vyr.pameti
0612 A952       0380         LDA #$52       ;cet sektor
0614 8D0203     0390         STA DCOMND     ;prikazovy byte
0617 A920       0400         LDA #$20       ;dolni b.cisla sektoru
0619 8D0A03     0410         STA DAUX1      ;  patri sem
061C A900       0420 LOOP    LDA #0         ;dolni byte adresy
061E 8D0403     0430         STA DBUFLO     ;  vyr. pameti
0621 2053E4     0440         JSR DSKINV     ;cti prvni sektor
0624 A980       0450         LDA #$80       ;posun o 128 bytu
0626 8D0403     0460         STA DBUFLO     ;  ve vyr.pameti
0629 EE0A03     0470         INC DAUX1      ;pristi sektor
062C AD0A03     0480         LDA DAUX1      ;uz jsme skoncili?
062F C960       0490         CMP #$60
0631 B010       0500         BCS FIN        ;ano
0633 2053E4     0510         JSR DSKINV     ;ne-cti dalsi sektor
0636 EE0503     0520         INC DBUFHI     ;zvys c.stranky
0639 EE0A03     0530         INC DAUX1      ;dalsi sektor
063C AD0A03     0540         LDA DAUX1      ;konec?
```

```
063F C960    0550        CMP  #$60
0641 90D9    0560        BCC  LOOP       ;ne
0643 60      0570 FIN    RTS             ;uplny konec
```

Jak vidíme, čím více se vzdálíme od CIO, o tím více věcí se musíme sami starat. V tomto programu musíme sami hlídat zvyšování čísla sektoru a adresy vyrovnávací paměti po každém čtení a musíme trvale testovat, zda jsme již skončili, porovnáváním čísla sektoru s číslem požadovaného posledního, $60. To je to, co jsme míníli přirovnáním jednotlivých typů ovládání vstupu a výstupu k slupkám cibule. Čím více se blížíme k jádru, tím máme více práce a tím méně pro nás dělá systém.

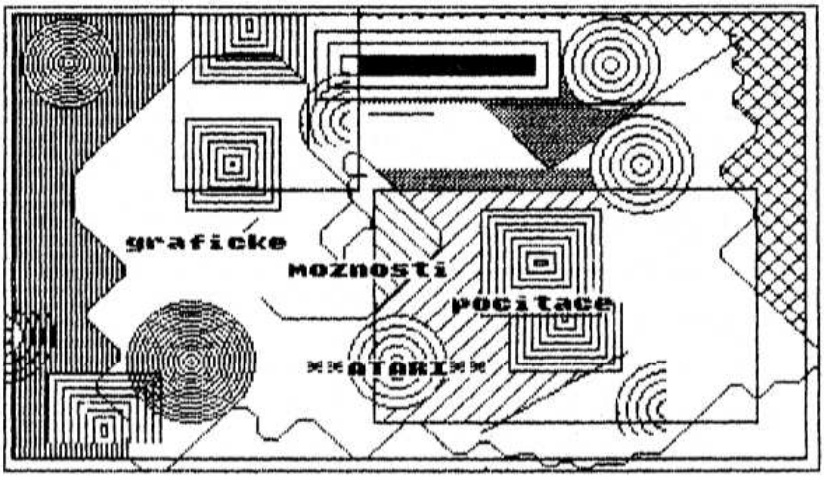
Úplně uvnitř je seriový vstupní a výstupní systém (SIO). V programu jsme použili DSKINV, ale namísto toho jsme mohli použít SIOV. V tom případě bychom však museli vytvořit celý DCB a nejen příslušné byty, jak jsme to udělali. Např. ID seriového busu by musel být dosazen na $31 v DDEVIC, status na 0 a hodnota čekacího intervalu (timeout) na nějakou rozumnou hodnotu, dejme tomu 45. Pak bychom mohli docílit stejného výsledku záměnou volání DSKINV voláním SIO (ovšem za cenu práce navíc).

Poznamenejme, že CIOV a DSKINV samy volají SIO pro vlastní provedení seriového přenosu, ale musejí zajistit nastavení veškeré potřebné informace před tímto voláním. Čím více se vzdalujete od CIO, tím více máte systém v ruce, ale také máte více práce. To platí při programování obecně - nejjednodušší je pro programování použít jazyk vysoké úrovně, tam ale máte nejmenší kontrolu nad systémem. Chcete-li získat větší kontrolu, musíte tvrději pracovat. Jistě jste však nečekali, že něco získáte bez úsilí, nebo ano?

Tím zakončíme naši diskusi o vstupu a výstupu na disku. Teď by vám mělo být zcela jasné, jak přenést informaci z disku a na disk při použití zřetězených i sekvenčních souborů. Pohrajte si s těmito způsoby až budete cítit, že je bezpečně ovládáte, protože tvoří základ mnoha aplikací, o které se určitě pokusíte.

Kapitola 10:

Grafika a zvuk v Assembleru



Grafika

Jednou z nejefektnějších a unikátních vlastností počítačů ATARI je jejich vynikající grafika. Při srovnání kvality grafiky s jinými známými mikropočítači je ATARI obecně jasný vítěz. Ve skutečnosti většina her, dostupných na několika různých počítačích, vypadá nejlépe na ATARI a reklama na ně většinou používá fotografie, získané ze stínítka ATARI.

"Ale", namítnete, "tu je možné používat pouze z BASICu." Uživatelé ATARI většinou nesprávně předpokládají, že grafické podprogramy jsou součástí BASICu a že příkazy jako PLOT a DRAWTO nemohou bez BASICu být použity. Ve skutečnosti jsou všechny grafické podprogramy součástí OS a jsou tudíž k dispozici z kteréhokoliv jazyka. Uvidíme teď, jak je používat z jazyka assembleru.

Každý program, který potřebuje příkazy jako GRAPHICS n, PLOT nebo jiné grafické příkazy, používá tyto grafické podprogramy mnohokrát. Je proto jednodušší si je připravit jako soubor podprogramů v assembleru, které mohou být zavolány z libovolného programu. Tyto podprogramy mohou být společně uloženy na disku a zahrnuty do libovolného programu, který je bude potřebovat. Pro použití těchto podprogramů v programu budete obecně potřebovat naplnit registry X,Y a akumulátor parametry a pak pomocí JSR podprogram zavolat. Toto předávání parametrů je popsáno v poznámkách ke každému podprogramu, aby jejich použití bylo zřejmé. Podrobný rozbor podprogramů bude v textu, následujícím za výpisem programu.

Grafické podprogramy

```
                0100 ; ****************************
                0110 ; symbolicke nazvy CIO
                0120 ; ****************************
0340            0130 ICHID  =    $0340
0341            0140 ICDNO  =    $0341
0342            0150 ICCOM  =    $0342
0343            0160 ICSTA  =    $0343
0344            0170 ICBAL  =    $0344
0345            0180 ICBAH  =    $0345
0346            0190 ICPTL  =    $0346
0347            0200 ICPTH  =    $0347
0348            0210 ICBLL  =    $0348
0349            0220 ICBLH  =    $0349
034A            0230 ICAX1  =    $034A
034B            0240 ICAX2  =    $034B
E456            0250 CIOV   =    $E456
                0260 ; ****************************
                0270 ; ostatni symbolicke nazvy
                0280 ; ****************************
02C4            0290 COLOR0 =    $02C4
0055            0300 COLCRS =    $55
0054            0310 ROWCRS =    $54
02FB            0320 ATACHR =    $02FB
00CC            0330 STORE1 =    $CC
00CD            0340 STOCOL =    $CD
```

```
0000            0350            *=   $600
                0360 ; ****************************
                0370 ; prikaz SETCOLOR
                0380 ; ****************************
                0390 ; Pred zavolanim podprogramu
                0400 ; musi byt v registrech hodnoty
                0410 ; obdobne jako v prikazu
                0420 ;    SETCOLOR reg,barva,jas
                0430 ; umistene po rade v registrech
                0440 ;    X, akumulator, Y
                0450 SETCOL
0600 0A         0460            ASL  A          ;nasobeni barvy 16
0601 0A         0470            ASL  A          ;  pomoci 4 volani
0602 0A         0480            ASL  A          ;  ASL
0603 0A         0490            ASL  A          ;  ted je barva*16
0604 85CC       0500            STA  STORE1     ;uschovej
0606 98         0510            TYA             ;preneseme jas
0607 18         0520            CLC             ;pred scitanim
0608 65CC       0530            ADC  STORE1     ;ted mame soucet
060A 9DC402     0540            STA  COLOR0,X   ;vlastni SETCOLOR
060D 60         0550            RTS             ;skoncili jsme
                0560 ; ****************************
                0570 ; prikaz COLOR
                0580 ; ****************************
                0590 ; v techto podprogramech
                0600 ; potrebujeme, aby hodnota n
                0610 ; bezne barvy byla ulozena
                0620 ; v akumulatoru; simulujeme
                0630 ;  SETCOLOR n
                0640 ;
                0650 COLOR
060E 85CD       0660            STA  STOCOL     ;to je vsechno!
0610 60         0670            RTS
                0680 ; ****************************
                0690 ; prikaz GRAPHICS
                0700 ; ****************************
                0710 ; parametr n prikazu
                0720 ; se predava tomuto podprogramu
                0730 ; se tomuto podprogramu
                0740 ; predava v akumulatoru
                0750 GRAFIC
0611 48         0760            PHA             ;uschovej v zasobniku
0612 A260       0770            LDX  #$60       ;IOCB6 pro stinitko
0614 A90C       0780            LDA  #$C        ;prikaz CLOSE
0616 9D4203     0790            STA  ICCOM,X    ;  do prikaz. bytu
0619 2056E4     0800            JSR  CIOV       ;uzavri soubor
061C A260       0810            LDX  #$60       ;opet stinitko
061E A903       0820            LDA  #3         ;prikaz OPEN
0620 9D4203     0830            STA  ICCOM,X    ;  do prikaz.bytu
0623 A9AD       0840            LDA  #NAME                255 ;jmeno je S:
0625 9D4403     0850            STA  ICBAL,X    ;  dolni byte
0628 A906       0860            LDA  #NAME/256  ; horni byte
062A 9D4503     0870            STA  ICBAH,X
062D 68         0880            PLA             ;vezmi GRAPHICS n
062E 9D4B03     0890            STA  ICAX2,X    ;graficky modus
0631 29F0       0900            AND  #$F0       ;horni 4 bity
0633 4910       0910            EOR  #$10       ;invertuj horni bit
0635 090C       0920            ORA  #$C        ;cteni nabo zapis
0637 9D4A03     0930            STA  ICAX1,X    ;n0,16,n0,32 atd.
```

```
063A 2056E4 0940         JSR  CIOV       ;proved GRAPHICS n
063D 60     0950         RTS             ;hotovo
            0960 ; ***************************
            0970 ; prikaz POSITION
            0980 ; ***************************
            0990 ; identicke prikazu
            1000 ; POSITION X,Y v BASICu
            1010 ; protoze X muze byt > 255
            1020 ; v modu GRAPHICS 8,
            1030 ; pouzijeme akumulator
            1040 ; pro horni byte X
            1050 POSITN
063E 8655   1060         STX  COLCRS     ;dolni byte X
0640 8556   1070         STA  COLCRS+1   ;horni byte X
0642 8454   1080         STY  ROWCRS     ;poloha Y
0644 60     1090         RTS             ;hotovo
            1100 ; ***************************
            1110 ; prikaz PLOT
            1120 ; ***************************
            1130 ; pouzijeme X,Y,A stejne jako
            1140 ; v prikaze POSITION
            1150 PLOT
0645 203E06 1160         JSR  POSITN     ;uloz informaci
0648 A260   1170         LDX  #$60       ;pro stinitko
064A A90B   1180         LDA  #$B        ;PUT vety
064C 9D4203 1190         STA  ICCOM,X    ;prikazovy byte
064F A900   1200         LDA  #0         ;specialni pripad
0651 9D4803 1210         STA  ICBLL,X    ;  s pouzitim
0654 9D4903 1220         STA  ICBLH,X    ;  akumulatoru
0657 A5CD   1230         LDA  STOCOL     ;vezmi barvu
0659 2056E4 1240         JSR  CIOV       ;nakresli bod
065C 60     1250         RTS             ;hotovo
            1260 ; ***************************
            1270 ; prikaz DRAWTO
            1280 ; ***************************
            1290 ; pouzijeme X,Y,A stejne jako
            1300 ; v prikaze POSITION
            1310 DRAWTO
065D 203E06 1320         JSR  POSITN     ;uloz informaci
0660 A5CD   1330         LDA  STOCOL     ;vezmi barvu
0662 8DFB02 1340         STA  ATACHR     ;pro CIO
0665 A260   1350         LDX  #$60       ;zase stinitko
0667 A911   1360         LDA  #$11       ;pro DRAWTO
0669 9D4203 1370         STA  ICCOM,X    ;prikazovy byte
066C A90C   1380         LDA  #$C        ;jako v XIO
066E 9D4A03 1390         STA  ICAX1,X    ;doplnkovy byte#1
0671 A900   1400         LDA  #0         ;vynuluj
0673 9D4B03 1410         STA  ICAX2,X    ;doplnkovy byte#2
0676 2056E4 1420         JSR  CIOV       ;nakresli caru
0679 60     1430         RTS             ;hotovo
            1440 ; ***************************
            1450 ; prikaz FILL
            1460 ; ***************************
            1470 ; pouzijeme X,Y,A stejne jako
            1480 ; prikaze POSITION  ,tento
            1490 ; je podobny DRAWTO
            1500 FILL
067A 203E06 1510         JSR  POSITN     ;uloz informaci
067D A5CD   1520         LDA  STOCOL     ;vezmi barvu
```

```
067F 8DFB02 1530         STA  ATACHR     ;pro CIO
0682 A960   1540         LDA  #$60       ;stinitko
0684 A912   1550         LDA  #$12       ;pro FILL
0686 9D4203 1560         STA  ICCOM,X    ;prikazovy byte
0689 A90C   1570         LDA  #$C        ;jako v XIO
068B 9D4A03 1580         STA  ICAX1,X    ;doplnkovy byte#1
068E A900   1590         LDA  #0         ;vynuluj
0690 9D4B03 1600         STA  ICAX2,X    ;doplnkovy byte#2
0693 2056E4 1610         JSR  CIOV       ;proved FILL
0696 60     1620         RTS             ;hotovo
            1630 ; ***************************
            1640 ; prikaz LOCATE
            1650 ; ***************************
            1660 ; pouzijeme X,Y,A jako
            1670 ; v POSITION
            1680 ; v akumulatoru bude
            1690 ; zjistena barva
            1700 LOCATE
0697 203E06 1710         JSR  POSITN     ;uloz informaci
069A A260   1720         LDX  #$60       ;stinitko
069C A907   1730         LDA  #7         ;GET vetu
069E 9D4203 1740         STA  ICCOM,X    ;prikazovy byte
06A1 A900   1750         LDA  #0         ;specialni pripad
06A3 9D4803 1760         STA  ICBLL,X    ;  prenosu info
06A6 8D4903 1770         STA  ICBLH      ;H ;  do akumulatoru
06A9 2056E4 1780         JSR  CIOV       ;proved LOCATE
06AC 60     1790         RTS             ;hotovo
            1800 ; ***************************
            1810 ; jmeno stinitka
            1820 ; ***************************
06AD 53     1830 NAME    .BYTE "S:",$9B
06AE 3A
06AF 9B
```

Popis podprogramů

První, co k těmto podprogramům musíme uvést je, že využívají běžné symbolické názvy z CIO a šest dalších. Nepotřebujeme zcela nový soubor symbolických názvů, protože využijeme standardní podprogramy CIO pro všechny grafické příkazy. Ze šesti nových symbolických názvů jsou dva názvy míst v paměti: STOCOL se používá k úschově informace o barvě v několika podprogramech a STORE1 se používá pro přechodné uchování informace. Byly pro ně zvoleny buňky $CD a $CC ale můžete je umístit kamkoliv do bezpečného místa v paměti, které si vyberete. Jedno takové místo by mohlo být $100 a $101, což jsou nejnižší 2 buňky zásobníku. Jiné z nových symbolických jmen je COLOR0, což je první z 5 míst paměti, používaných pro uchovávání informace o barvě v počítačích ATARI na buňkách 708 až 712. Druhé je COLCRS, 2 bytové místo v paměti na $55 a $56, v nichž je vždy uloženo číslo sloupce, ve kterém se právě nachází kursor. Protože v modu GRAPHICS 8 je celkem 320 sloupců, potřebujeme 2 byty pro uložení všech možných poloh kursoru. Ve všech jiných grafických modech však je jasné, že na $56 bude vždy nula. Třetí nový symbolický název je ROWCRS v buňce $54, v němž je uložena vertikální poloha kursoru. Žádný grafický modus nemá více než 192 vertikálních poloh, takže pro uschování této informace stačí 1 byte.

Poslední z nových symbolických názvů je ATACHR na $2FB, kde je uložena informace o barvě čáry, kreslené ve FILL a DRAWTO.

Podprogramy byly přeloženy s použitím počátku na $600. Pokud plánujete jejich použití ve větším programu v assembleru, stačí, když je přečíslujete na nějaká vysoká čísla řádků jako 25000 a výše a zařadíte je do svého programu před překladem. Takto budete mít k dispozici všechny grafické příkazy z assembleru bez nutnosti je pracně zadávat do každého programu, který napíšete.

První podprogram je ekvivalent příkazu SETCOLOR v BASICu. Víme, že má standardní tvar:

```
SETCOLOR bar,registr,barva,jas
```

V podprogramu v assembleru musíme nejprve naplnit registry 6502 odpovídající informací. Typické volání, které simuluje příkaz BASICu SETCOLOR 2,4,10 bude vypadat takto:

```
25  LDX #2
30  LDA #4
35  LDY #10
40  JSR SETCOL
```

Použili jsme pro název šest písmen jména SETCOLOR, takže tento podprogram může být použit v libovolném z dostupných assemblerů na ATARI. Pokud vámi používaný assembler dovoluje používání jmen delších než 6 písmen, klidně použijte pro podprogram celé jméno. Podobným způsobem jsou utvořena i jména dalších podprogramů – např. POSITN za POSITION.

K provedení příkazu SETCOLOR potřebujeme přičíst jas k 16ti násobku hodnoty barvy a výsledek uložit do příslušného barvového registru. K vytvoření násobku 16 použijeme akumulátor a provedeme 4 instrukce ASL A. Protože každá z nich zdvojnásobí obsah akumulátoru, bude výsledkem 16ti násobek původní hodnoty. Po vynásobení uložíme výsledek do pomocné buňky a do akumulátoru přeneseme instrukcí TYA hodnotu jasu jako přípravu pro sčítání. Musíme pak vynulovat bit přenosu C jako obvykle před sčítáním a k jasu přičteme výsledek předchozího násobení. Konečně použijeme registr X, v němž je číslo požadovaného barvového registru, a použijeme jej jako index do tabulky 5 barvových registrů, o nichž byla řeč výše. Protože jsme chtěli udělat SETCOLOR 2, naplníme před voláním podprogramu 2 do registru X a příslušná informace bude uložena do $2C6.

Další podprogram, příkaz COLOR, je zdaleka nejjednodušší ze všech. K provedení ekvivalentu příkazu v BASICu COLOR 3 potřebujeme následující kód:

```
25  LDA #3
30  JSR COLOR
```

Tento podprogram jednoduše uloží vybranou barvu do naší buňky v paměti STOCOL, kde bude k dispozici pro další grafické podprogramy.

Příkaz GRAPHICS je implementován podobně. K napodobení příkazu v BASICu GRAPHICS 23 použijeme následující kód:

```
25  LDA #23
30  JSR GRAFIC
```

První, co musíme udělat, je přechodné uložení hodnoty grafického modu. Mohli bychom ji uložit do STORE1, ale v tomto případě je rychlejší uložení do zásobníku; tentokrát ji nebudeme násobit ani přičítat jako v SETCOLOR. Další 4 řádky uzavřou obrazovku jako zařízení. To je pro jistotu. Pokud je obrazovka již uzavřena, nic tím

nezkazíme. Avšak je-li otevřena a pokusíme se ji znovu otevřít, dojde k chybě, takže ji raději pro jistotu uzavřeme. Použijeme IOCB6 (tím, že do registru X uložíme $60) a tím specifikujeme obrazovku, které je IOCB6 v ATARI standardně přiřazen.

Zbytek příkazu GRAPHICS otevře obrazovku v grafickém modu, který požadujeme. Opět použijeme IOCB6, do příkazového bytu uložíme příkaz OPEN v řádku 830. Jméno stínítka obrazovky je S: a adresu tohoto jména uložíme do ICBAL a ICBAH. Grafický modus se pak vyzvedne ze zásobníku a uloží do druhého doplňkového bytu. Důležité bity v ICAX2 jsou dolní 4, které specifikují vlastní grafický modus, v tomto případě GRAPHICS 7. Horní 4 bity řídí výmaz stínítka, přítomnost textového okénka a podobně, jak bylo popsáno v kapitole 8. V tomto případě jsme ke grafickému modu přidali 16, abychom vyřadili textové okénko. Pro vytažení těchto bitů provedeme AND s hodnotou $F0, které nám získá horní půlbyte (nibble) grafického modu. OS požaduje, aby horní bit této informace byl invertován, takže dále provedeme EOR s $10 k překlopení tohoto bitu. Konečně doplníme dolní půlbyte $C, abychom umožnili zápis nebo čtení stínítka a byte uložíme do ICAX1 bloku IOCB. Grafický podprogram končí voláním CIO, které nám nastaví obrazovku na požadované hodnoty.

Jak už jsme si řekli, příkaz POSITION pro GRAPHICS 8 vyžaduje 320 poloh X, takže potřebujeme 2 byty pro uložení tohoto velkého čísla. Proto pro simulaci příkazu POSITION 285,73 uložíme dolní byte souřadnice X do registru X, horní byte do akumulátoru a souřadnici Y do registru Y:

```
25  LDX #30
30  LDA #1
35  LDY #73
40  JSR POSITN
```

Zřejmě v každém jiném modu než GRAPHICS 8 bude před voláním POSITN do akumulátoru uložena nula a registr X bude obsahovat souřadnici X. Podprogram vždy uloží odpovídající informaci do příslušných míst v paměti. Souřadnice X se uloží do COLCRS a COLCRS+1 a souřadnice Y do ROWCRS.

Příkaz PLOT v BASICu: PLOT 258,13 je v assembleru nasimulován následovně:

```
25  LDX #3
30  LDA #1
35  LDY #13
40  JSR PLOT
```

Přitom se použije téže konvence jako pro příkaz POSITN. Ve skutečnosti podprogram PLOT začíná voláním podprogramu POSITN, která uloží předanou informaci na správná místa pro použití OS po zavolání CIO. Protože chceme výstup na stínítko, použijeme IOCB6 a příkazový byte je $B pro PUT věty. V tomto případě chceme vyslat jediný byte informace a proto použijeme speciální volání s hodnotou délky rovnou nule a byte se vezme z akumulátoru. Do akumulátoru naplníme informaci o barvě a zavolání CIO nám nakreslí požadovaný bod.

Podprogramy DRAWTO a FILL jsou si tak podobné, že je popíšeme společně. Volací posloupnost je pro oba příkazy identická, takže pro simulaci příkazu v BASICu DRAWTO 42,80 použijeme instrukce

```
25  LDX #42
30  LDA #0
35  LDY #80
```

```
40  JSR DRAWTO
```

Pro příkaz FILL změňte řádek 40 na JSR FILL.

Podprogram začíná zavoláním POSITN pro uložení předávané informace. Informace o barvě je pak uložena do ATACHR a použijeme opět ICCB6, příkazový byte naplníme $11 pro DRAWTO a $12 pro FILL. ICAX1 potřebuje $C a před zavoláním CIO ještě vynulujeme ICAX2. Tyto podprogramy přesně odpovídají příslušné instrukci XIO v BASICu, která udělá totéž. Např. pro nakreslení čáry bychom mohli použít příkaz:

```
XIO 17,#6,12,0,"S:"
```

V tomto příkaze je 17 příkazový byte $11, #6 je číslo IOCB, 12 se uloží do ICAX1, nula do ICAX2 a jméno zařízení je "S:". Pro příkaz FILL může být opět použita stejná instrukce jednoduše záměnou 17 za =8 ($12).

Poslední podprogram pro příkaz LOCATE je prakticky totožný s příkazem PLOT s tím rozdílem,že použijeme příkaz GET namísto PUT. Opět je použit speciální případ vstupu s ICBLL a ICBLH nastavenými na 0. Volací posloupost pro simulaci příkazu v BASICu LOCATE 10,12,A je následující:

```
25  LDX #10
30  LDA #0
35  LDY #12
40  JSR LOCATE
```

V tomto případě bude po návratu akumulátor obsahovat hodnotu barvové informace na souřadnicích 10,12. Použitím STA ji můžeme uchovat; případně může být použita přímo pro porovnávání s požadovanou hodnotou nebo jakýmkoliv jiným způsobem.

Tím končíme diskusi o assemblerových protějších grafických příkazů v BASICu. Použijte je v nějakém jednoduchém programu a sami uvidíte, jak rychle vám přejdou do krve a jak jsou snadné. Ve skutečnosti se používají téměř stejně snadno jako příkazy v BASICu. Protože však jak BASIC tak Assembler používají tytéž podprogramy OS pro příkazy DRAWTO a FILL, nečekejte, že podprogramy v assembleru budou o moc rychlejší než ty, na které jste zvyklí z BASICu. Budou o něco rychlejší, protože nemusíte platit ztrátové časy interpretace v BASICu, ale nikde neuvidíte ten rozdíl v rychlosti, který jste si zvykli očekávat při přechodu z BASICu k Assembleru. Abyste dosáhli urychlení i zde, museli byste napsat vlastní podprogramy DRAWTO a FILL s úplně jinou logikou než ta, která je použita v ATARI OS. Takové podprogramy byly již napsány a jsou mnohem rychlejší než podprogramy OS, ale nejsou obecně dostupné, takže v případě potřeby si musíte napsat vlastní.

Když teď nabíráte zkušenost v assembleru, mohli byste chtít změnit pro své účely i ústřední podprogramy ATARI. Chcete-li se o to pokusit, velmi vám doporučuji, abyste si opatřili od ATARI výpisy OS a Technický manuál (Technical User s Notes). Můžete si pak prohlédnout komentovaný zdrojový text podprogramů OS a pozměnit jej pro vlastní účely. Jednoduše je zařaďte jako část vlastních programů a udělejte změny podle libosti. Nezpomeňte však, že kód OS náleží ATARI. Takové modifikace můžete používat ve svých programech pro vlastní potřebu, ale chcete-li nabídnout k prodeji libovolný program, obsahující části ATARI OS, vyžádejte si svolení od ATARI.

Jedna snadná změna je dovolit kreslení bez testování polohy kursoru mimo stínítko, což věc poněkud zpomaluje. Jen si musíte být jisti, že váš program počítá hodnoty správně - jinak...

Uvědomte si, že vše, co je možné z BASICu, je rovněž možné z assembleru. Často uváděný příklad je oživení obrázků pomocí rotace barvových registrů obrázku s použitím speciálních modů GTIA nebo běžných grafických modů. Velmi jednoduchý podprogram může cyklicky zaměnit hodnoty standardních barvových registrů téměř okamžitě:

```
15  LDA  708
20  STA STOCOL
25  LDA  709
30  STA  708
35  LDA  710
40  STA  709
45  LDA  711
50  STA  710
55  LDA  712
60  STA  711
65  LDA STOCOL
70  STA  712
```

Když nyní umíte kreslit detailní obrázky v assembleru, můžete tento trik použít k oživení obrázků téměř bez zpomalení v běhu programu. Např. implementace běhu příkopem je jednoduchá, vytvořením iluze běhu pomocí rotace barev.

Player-missile graphics - PMG

Jinou velice zajímavou vlastností počítačů ATARI je grafika hráčů a střel. Už jsme viděli příklad na použití podprogramu v assembleru k pohybu hráče. V programu bylo celé nastavení PMG v BASICu a pouze podprogram pro pohyb hráče byl v assembleru. Abychom ukázali, jak provést tyto operace v programu napsaném pouze v assembleru, byl program zcela přepsán do assembleru a je zde uveden. Většina adres v programu je uvedena dekadicky, protože takto byly uvedeny v BASICu. Program je tak podobný programu v BASICu jak jen bylo možné.

```
                0100 ; ****************************
                0110 ; symbolicke nazvy CIO
                0120 ; ****************************
0340            0130 ICHID  =    $0340
0341            0140 ICDNO  =    $0341
0342            0150 ICCOM  =    $0342
0343            0160 ICSTA  =    $0343
0344            0170 ICBAL  =    $0344
0345            0180 ICBAH  =    $0345
0346            0190 ICPTL  =    $0346
0347            0200 ICPTH  =    $0347
0348            0210 ICBLL  =    $0348
0349            0220 ICBLH  =    $0349
034A            0230 ICAX1  =    $034A
034B            0240 ICAX2  =    $034B
E456            0250 CIOV   =    $E456
                0260 ; ****************************
                0270 ; ostatni symbolicke nazvy
                0280 ; ****************************
00CC            0290 YLOC   =    $CC        ;neprima adr. pro Y
00CE            0300 XLOC   =    $CE        ;pro uchovani pol.X
00D0            0310 INITX  =    $D0        ;poc.adresa X
00D1            0320 INITY  =    $D1        ;poc.adresa Y
```

```
0100            0330 STOTOP =    $100       ;pom. pamet
D300            0340 STICK  =    $D300      ;hw adr. STICK(0)
D000            0350 HPOSP0 =    $D000      ;hor.pol.P0
0000            0360         *=  $600
                0370 ; ****************************
                0380 ; snizeni adresy vrsku pameti
                0390 ; ****************************
0600 A56A       0400         LDA 106        ;vezmi vrsek pameti
0602 8D0001     0410         STA STOTOP     ;prechodne uloz
0605 38         0420         SEC            ;priprav odecitani
0606 E908       0430         SBC #8         ;vysetri 8 stranek
0608 856A       0440         STA 106        ;zapis novy vrsek pam.
060A 8D07D4     0450         STA 54279      ;PMBASE
060D 85CF       0460         STA XLOC+1     ;priprav neprimou
060F A900       0470         LDA #0         ;  adresu pro vymaz
0611 85CE       0480         STA XLOC       ;  pameti pro PMG
                0490 ; ****************************
                0500 ; nove vytvor GRAPHICS 0
                0510 ; ****************************
0613 A900       0520         LDA #0         ;GRAPHICS 0
0615 48         0530         PHA            ;uloz do zasobniku
0616 A260       0540         LDX #$60       ;IOCB6 pro stinitko
0618 A90C       0550         LDA #$C        ;prikaz CLOSE
061A 9D4203     0560         STA ICCOM,X    ;  do prikaz. bytu
061D 2056E4     0570         JSR CIOV       ;proved CLOSE
0620 A260       0580         LDX #$60       ;opet stinitko
0622 A903       0590         LDA #3         ;prikaz OPEN
0624 9D4203     0600         STA ICCOM,X    ;prikaz OPEN
0627 A9ED       0610         LDA #NAME                255 ;jmeno je "S:"
0629 9D4403     0620         STA ICBAL,X    ;  dolni byte
062C A906       0630         LDA #NAME/256  ;  horni byte
062E 9D4503     0640         STA ICBAH,X
0631 68         0650         PLA            ;vezmi GRAPHICS 0
0632 9D4B03     0660         STA ICAX2,X    ;graficky modus
0635 29F0       0670         AND #$F0       ;horni 4 bity
0637 4910       0680         EOR #$10       ;prepni horni bit
0639 090C       0690         ORA #$C        ;zapis nebo cteni
063B 9D4A03     0700         STA ICAX1,X    ;n+16,n+32 atd.
063E 2056E4     0710         JSR CIOV       ;nastav GRAPHICS 0
                0720 ; ****************************
                0730 ; nastav PMG
                0740 ; ****************************
0641 A978       0750         LDA #120       ;pocatecni hodnota X
0643 85D0       0760         STA INITX      ;  uloz na misto
0645 A932       0770         LDA #50        ;pocatecni hodnota Y
0647 85D1       0780         STA INITY      ;  uloz na misto
0649 A92E       0790         LDA #46        ;rozliseni je
064B 8D2F02     0800         STA 559        ;  dva radky
                0810 ; ****************************
                0820 ; vynuluj cast pameti pro PM
                0830 ; ****************************
064E A000       0840         LDY #0         ;pouzij jako citac
0650 A900       0850         LDA #0         ;bude se ukladat
                0860 CLEAR
0652 91CE       0870         STA (XLOC),Y   ;nuluj byte
0654 88         0880         DEY            ;je konec stranky?
0655 D0FB       0890         BNE CLEAR      ;ne, jeste pokracuj
0657 E6CF       0900         INC XLOC+1     ;stranka hotova
0659 A5CF       0910         LDA XLOC+1     ;na dalsi stranku
```

```
065B CD0001 0920         CMP  STOTOP     ;skoncili jsme?
065E F0F2   0930         BEQ  CLEAR      ;jeste dalsi stranku
0660 90F0   0940         BCC  CLEAR      ;pokracuj
            0950 ; ***************************
            0960 ; nyni ulozime hrace do
            0970 ; prislusneho mista v pameti
            0980 ; vyhrazene pro PMG
            0990 ; ***************************
0662 A56A   1000         LDA  106        ;nejdrive spocitej
0664 18     1010         CLC             ;  spravnou polohu Y
0665 6902   1020         ADC  #2         ;PMBASE+512 (2 stranky)
0667 85CD   1030         STA  YLOC+1     ;horni byte YLOC
0669 A5D1   1040         LDA  INITY      ;pridej souradnici Y
066B 85CC   1050         STA  YLOC       ;  do dolniho bytu
066D A000   1060         LDY  #0         ;pouzij jako citac
            1070 INSERT
066F B9F006 1080         LDA  PLAYER,Y   ;vezmi byte hrace
0672 91CC   1090         STA  (YLOC),Y   ;uloz na misto
0674 C8     1100         INY             ;pro dalsi byte
0675 C008   1110         CPY  #8         ;skoncili jsme?
0677 D0F6   1120         BNE  INSERT     ;jeste ne
0679 A5D0   1130         LDA  INITX      ;vezmi pocatecni X
067B 8D00D0 1140         STA  53248      ;predej ATARI
067E 85CE   1150         STA  XLOC       ;i sem
0680 A944   1160         LDA  #68        ;cervena barva hrace
0682 8DC002 1170         STA  704        ;  jako v BASICu
0685 A903   1180         LDA  #3         ;povoleni PMG (player
0687 8D1DD0 1190         STA  53277      ;  missile graphics)
            1200 ; ***************************
            1210 ; kratka hlavni smycka
            1220 ; ***************************
            1230 MAIN
068A 209A06 1240         JSR  RDSTK      ;cti joystick
068D A205   1250         LDX  #5         ;pro rizeni hrace
068F A000   1260         LDY  #0         ;  musime pridat
            1270 DELAY
0691 88     1280         DEY             ;  zpozdeni
0692 D0FD   1290         BNE  DELAY      ;  abychom program
0694 CA     1300         DEX             ;  trochu zpomalili
0695 D0FA   1310         BNE  DELAY
0697 4C8A06 1320         JMP  MAIN       ;a vse zopakujeme
            1330 ; ***************************
            1340 ; cteni joysticku #1
            1350 ; ***************************
            1360 RDSTK
069A AD00D3 1370         LDA  STICK      ;cti pol.joysticku
069D 2901   1380         AND  #1         ;testuj bit 0
069F F016   1390         BEQ  UP         ;je 0,poloha 11,12,1h
06A1 AD00D3 1400         LDA  STICK      ;cti znovu
06A4 2902   1410         AND  #2         ;testuj bit 1
06A6 F020   1420         BEQ  DOWN       ;je 0,pol. 5,6,7 h.
06A8 AD00D3 1430 SIDE    LDA  STICK      ;jeste precti
06AB 2904   1440         AND  #4         ;testuj bit 3
06AD F02E   1450         BEQ  LEFT       ;je 0,pol. 8,9,10 h.
06AF AD00D3 1460         LDA  STICK      ;a znovu cti
06B2 2908   1470         AND  #8         ;testuj bit 4
06B4 F02F   1480         BEQ  RIGHT      ;je 0,pol. 2,3,4 h.
06B6 60     1490         RTS             ;joyst.v zakladni poloze
            1500 ; ***************************
```

```
                1510 ; posun hrace prisl. smerem
                1520 ; pocinaje pohybem nahoru
                1530 ; ****************************
06B7 A001       1540 UP      LDY  #1          ;priprava pro byte 1
06B9 C6CC       1550         DEC  YLOC        ;zmensi o 1
06BB B1CC       1560 UP1     LDA  (YLOC),Y    ;ber prvni bytw
06BD 88         1570         DEY              ;pro posuv o 1 posici
06BE 91CC       1580         STA  (YLOC),Y    ;vlastni presun
06C0 C8         1590         INY              ;puvodni hodnota
06C1 C8         1600         INY              ;pro pristi byte
06C2 C00A       1610         CPY  #10         ;skoncili jsme?
06C4 90F5       1620         BCC  UP1         ; jeste ne
06C6 B0E0       1630         BCS  SIDE        ;vynuceny skok
                1640 ; ****************************
                1650 ; posun hrace dolu
                1660 ; ****************************
06C8 A007       1670 DOWN    LDY  #7          ;nejprve horni byte
06CA B1CC       1680 DOWN1   LDA  (YLOC),Y    ;vezmi horni
06CC C8         1690         INY              ;pro posun dolu
06CD 91CC       1700         STA  (YLOC),Y    ;presun byte
06CF 88         1710         DEY              ;zpet na poc. hodnotu
06D0 88         1720         DEY              ;dalsi nizsi byte
06D1 10F7       1730         BPL  DOWN1       ;pokud Y>= pokracuj
06D3 C8         1740         INY              ;nastav na 0
06D4 A900       1750         LDA  #0          ;pro vynulovani hor.b.
06D6 91CC       1760         STA  (YLOC),Y    ;vynuluj jej
06D8 E6CC       1770         INC  YLOC        ;hrac je o 1 vyse
06DA 18         1780         CLC              pripav nuceny skok
06DB 90CB       1790         BCC  SIDE        ;vynuceny skok
                1800 ; ****************************
                1810 ; posuv na stranu, nejdr.vlevo
                1820 ; ****************************
06DD C6CE       1830 LEFT    DEC  XLOC        ;pro posuv doleva
06DF A5CE       1840         LDA  XLOC        ;vezmi polohu
06E1 8D00D0     1850         STA  HPOSP0      ;posunuti
06E4 60         1860         RTS              ;zpet do MAIN -hotovo
                1870 ; ****************************
                1880 ; posuv doprava
                1890 ; ****************************
06E5 E6CE       1900 RIGHT   INC  XLOC        ;pro pos. doprava
06E7 A5CE       1910         LDA  XLOC        ;vezmi jej
06E9 8D00D0     1920         STA  HPOSP0      ;posunuti
06EC 60         1930         RTS              ;hotovo - zpet do MAIN
                1940 ; ****************************
                1950 ; data
                1960 ; ****************************
06ED 53         1970 NAME    .BYTE "S:",$9B
06EE 3A
06EF 9B
06F0 FF         1980 PLAYER .BYTE 255,129,129,129,129,129,129,255
06F1 81
06F2 81
06F3 81
06F4 81
06F5 81
06F6 81
06F7 FF
```

Tento program obsahuje řadu podprogramů, o kterých jsme již

mluvili: podprogram k vynulování oblasti paměti pro PMG, čtení ovladače (Josticku) a posuv hráče, instrukci GRAPHICS 0. Zde jsme je jednoduše dali všechny dohromady do jednoho velkého programu, který provede všechny úkoly nutné pro implementaci jednoduchého příkladu na grafiku hráčů a střel v assembleru.

Analogicky k programu v BASICu, který jsme již napsali, verze v assembleru začíná snížením konce paměti o 8 stránek, aby se udělalo místo pro PMG. Řádky 400 až 440 vykonají tuto práci; řádek 410 uchová starou hodnotu RAMTOP pro další použití. Řádek 450 sdělí ATARI polohu PMBASE, novou hodnotu RAMTOP. Použijeme XLOC a XLOC+1 jako přechodnou nepřímou adresu ve stránce 0 pro použití v podprogramu pro vynulování paměti. Řádky 520 až 710 přenesou obraz stínítka GRAPHICS 0 pod novou polohu RAMTOP. Řádky 750 až 780 uloží počáteční hodnoty souřadnic X a Y, kde chceme, aby se hráč objevil. Hodnoty budou použity později a tyto řádky jsou zde pouze pro udržení analogie s programem v BASICu. Hodnoty bychom vůbec nemuseli ukládat, stejně snadno bychom je mohli uvést později přímo v programu. Kterékoliv z těchto způsobů pracuje, ale je-li program napsán takto, je o něco snazší do něj později dělat změny. Řádky 790 a 800 nastaví dvouřádkové rozlišení a v řádcích 840 až 940 pak vynulujeme celou oblast PMG.

V programu v BASICu určíme adresu v paměti, kam máme uložit hráče, aby se objevil na správném místě na stínítku takto:

PMBASE+512+INITY

Víme, že 512 bytů od PMBASE odpovídá 2 stránkám, protože každá stránka má 256 bytů. Tudíž víme, že horní byte této adresy v našem programu v assembleru musí být o 2 větší než PMBASE. V řádcích 1000 až 1030 vezmeme PMBASE, přičteme 2 a uložíme výsledek do YLOC+1, horního bytu polohy Y v paměti. Dolní byte je jednoduše INITY, počáteční poloha Y. Uvědomte si, že čím víc dolů na stínítku, tím výše v paměti musí být hráč uložen.

Pro uložení hráče do správného místa v paměti čteme postupně po jednom bytu z tabulky nazvané PLAYER a ukládáme je do paměti s použitím nepřímého adresování, které jsme předem připravili. Když je registr Y, náš čítač, roven 8, skončili jsme, protože jsme začali v nule a měli jsme přenést pouze 8 bytů. Kdyby váš hráč byl delší, jednoduše byste museli změnit jediný byte v řádku 1110 na hodnotu o 1 vyšší než počet bytů v hráči. Počáteční hodnota souřadnice X hráče se přečte z INITX a uloží do registru horizontální polohy hráče nula, 53248. Rovněž ji uložíme do XLOC pro použití v podprogramu pohybu hráče.

Pak nastavíme barvu hráče na červenou uložením čísla 68 do barvového registru pro hráče nula, 704, a povolíme zobrazení grafiky hráčů a střel uložením 3 do 53277, GRACTL.

Hlavní smyčka programu je jednoduchost sama. Uděláme JSR do podprogramu, který čte ovladač a posouvá hráče a pak vstoupíme do krátké zpožďovací smyčky. Kdybychom ji vynechali, hráč by se pohyboval tak rychle, že bychom jej vůbec nedokázali řídit! Pak se jednoduše vrátíme zpět na čtení ovladače a posun hráče.

Pokud byste chtěli programu přidat na zajímavosti, můžete samozřejmě vložit do této hlavní smyčky vlastní program pro zjištění kolize mezi hráčem a pozadím, můžete vytvářet překážky nebo cokoliv jiného. Pokud ale chcete svůj proram prodlužovat, měli byste změnit počátek a umístit program někam výše do paměti. Tak jak je, program zabírá téměř celou stránku 6, takže při jeho zvětšování bez změny počátku brzy začnete přepisovat DOS a nebudete schopni program uschovat nebo nahrát. Stačí pouze změnit počátek na $6000 nebo jiné bezpečné místo výše v paměti. K otestování programu po jeho přeložení

napište BUG pro přechod do ladicího modu (debugger) a napište G600 pro původní verzi (nebo G6000, pokud jste změnili začátek).

Když teď víte, jak implementovat grafiku hráčů a střel v assembleru, měli byste být schopni psát programy, používající stejnou techniku. Jak nám už ukázala nutnost zařazení zpožďovací smyčky do právě popsaného programu, výrazně tím vše zrychlíte a zajistíte plynulý pohyb hráčů pro výrazné zlepšení svých her. Mnoho úspěchů!

Podprogram pro zvuk (SOUND)

Vytváření zvuku na ATARI v BASICu je velice jednoduché, protože příkazem SOUND můžeme zapnout nebo vypnout libovolný z hlasů s libovolným zkreslením, kmitočtem i hlasitostí. Tytéž funkce jsou k dispozici z Assembleru. Můžeme napsat podprogram, který imituje činnost příkazu SOUND v BASICu, podobně jako jsme to udělali s grafickými příkazy v minulé sekci.

```
                0100 ; ******************************
                0110 ; symbolicke nazvy pro zvuk
                0120 ; ******************************
D200            0130 AUDF1  =    $D200      ;kmitocet kan.1
D201            0140 AUDC1  =    $D201      ;rizeni kan.1
D208            0150 AUDCTL =    $D208      ;rizeni zvuku
D20F            0160 SKCTL  =    $D20F      ;rizeni ser.portu
0101            0170 STORE2 =    $101       ;pomocna pamet
0000            0180         *=  $600
                0190 ; ******************************
                0200 ; prikaz SOUND
                0210 ; ******************************
                0220 ; pred zavolanim tohoto
                0230 ; podprogramu musi registr X
                0240 ; obsahovat pozadovane cislo
                0250 ; hlasu, akumulator hodnotu
                0260 ; zkresleni, registr Y vysku
                0270 ; a STORE2 ma obsahovat
                0280 ; pozadovanou vysku tonu
                0290 ;
                0300 SOUND
0600 48         0310         PHA             ;uchovej zkresleni
0601 8A         0320         TXA             zdvoj cislo hlasu
0602 0A         0330         ASL  A          ;  pro ofset na ridici
0603 AA         0340         TAX             ;  registr zvuku
0604 AD0101     0350         LDA  STORE2     ;kmitocet dej do
0607 9D00D2     0360         STA  AUDF1,X    ;  spravneho reg.
060A 8C0101     0370         STY  STORE2     ;pro pozdejsi pouz.
060D A900       0380         LDA  #0         ;inicializace cipu
060F 8D08D2     0390         STA  AUDCTL     ;  POKEY
0612 A903       0400         LDA  #3
0614 8D0FD2     0410         STA  SKCTL
0617 68         0420         PLA             ;vezmi zkresleni
0618 0A         0430         ASL  A          ;nasobeni 16
0619 0A         0440         ASL  A          ;  pro posun
061A 0A         0450         ASL  A          ;  zkresleni do horni
061B 0A         0460         ASL  A          ;  poloviny bytu
061C 18         0470         CLC             ;priprav scitani
061D 6D0101     0480         ADC  STORE2     ;pricti hlasitost
0620 9D01D2     0490         STA  AUDC1,X    ;do spravneho hlasu
0623 60         0500         RTS             ;a to je vse
```

V programu nejprve zdvojíme hodnotu, původně uloženou v registru X, která určuje použitý hlas. Je to proto, že zvukové registry jsou uspořádány po párech, takže kmitočtový registr a řídicí registr zabírají dohromady 2 byty v paměti. Kmitočet se vezme z buňky STORE2, kam jsme jej uložili, protože pro SOUND potřebujeme 4 parametry a nestačí nám registry X, Y a akumulátor. Kmitočet se pak v řádku 360 uloží do příslušného kmitočtového registru a přechodně si uložíme hlasitost do té doby než ji budeme potřebovat. V řádcích 380 až 410 inicializujeme POKEY.Pak přemístíme zkreslení do horní poloviny bytu v akumulátoru a přičteme hlasitost, abychom dostali hodnotu, kterou musíme uložit do příslušného řídicího registru. A to je vše, co bylo potřeba udělat!

Generování zvuku v Assembleru trpí stejným problémem jako v BASICu: není možné specifikovat délku vytvářeného zvuku. Jak příkaz SOUND v BASICu tak náš ekvivalentní podprogram v Assembleru pouze zvuk nastartují. Po určité době musíme zase zvuk vypnout, abychom vytvořili požadovaný zvuk nebo tón. K vypnutí zvuku stačí uložit nulu do příslušného řídicího registru AUDC1-4. Ke změření uběhlého času použijeme buď časomíru na buňkách 18 až 20 nebo program podobný tomu v grafice hráčů a střel pro krátké zpoždění:

```
      LDX #50
      LDY #0
LOOP  DEY
      BNE LOOP
      DEX
      BNE LOOP
```

Časové zpoždění zde je určeno počáteční hodnotou v registru X; čím větší číslo, tím delší zpoždění. Tyto dvě vnořené smyčky by trvaly věčnost, pokud bychom je naprogramovali v BASICu, ale v Assembleru je zpoždění velice krátké. Vyzkoušejte si, jak rychle se dokáže provést 256x50 tj. 12 800 smyček.

Odečítací časomíra (countdown timers)

Třetí způsob, jak měřit čas na ATARI, je s použitím odečítací časomíry. Jako časomíra mohou sloužit AUDF1, AUDF2 a AUDF4. Uloží-li se do STIMER ($D209) jakákoliv nenulová hodnota, hodnoty uložené v AUDF1, AUDF2 a AUDF4 se začnou zmenšovat. Každý z těchto tří registrů má odpovídající vektor na uvedených adresách:

Časovač	Vektor
AUDF1	$210,$211 (VTIMR1)
AUDF2	$212,$213 (VTIMR2)
AUDF3	$214,$215 (VTIMR4)

Do některého z těchto míst uložíme adresu krátkého programu, který vypne zvuk. Jakmile příslušný čítač projde nulou, vyrobí se přerušení a řízení se předá vašemu programu, který zvuk ukončí. Jen nezapomeňte ukončit tento program instrukcí RTI a ne RTS. Před použitím tohoto typu časování musíte uložit správnou hodnotu do bytu povolení přerušení IRQEN ($D20E). Pro povolení VTIMR1, dejte jedničku do bitu 0 bytu IRQEN; pro VTIMR2 dosaďte bit 1 a pro VTIMR4 bit 2 bytu IRQEN.

Teď byste měli být schopni vyrobit jakýkoliv zvuk, dostupný z

BASICu, také v asembleru. Tady je jeden, který v BASICu vyrobeň být nemůže, kvůli rychlosti. Jestliže je uložena jednička do bitu 4 libovolného z řídicích registrů zvuku, zaslechnete krátké lupnutí. To je způsobeno jedním povytažením membrány reproduktoru. Pokud budeme rychle střídat 0 a 1 v tomto bitu, můžeme vytvářet zvuk přímo kmitáním membrány. Zkuste toto:

```
LOOP LDA #16
     STA AUDC1
     (vložte zpoždění pro kmitočet)
     LDA #31
     STA AUDC1
     (vložte zpoždění pro kmitočet)
     JMP LOOP
```

Membrána reproduktoru vašeho televizoru nebo monitoru bude vibrovat tam a zpět, vytvářejíc zvuk úplně jiným způsobem než bylo popsáno výše. V tomto případě pro vytváření zvuku pohybujeme přímo membránou.

D:GRAMO

Příloha 1:

Instrukční soubor 6502

V této příloze budou popsány všechny instrukce procesoru 6502. Jejich zápis závisí od použitého assembleru a může se v syntaktických detailech někdy lišit.

Společně s každou instrukcí, které budou uvedeny abecedně, najdete zároveň:

1. Příklady použití
2. Popis použitých adresních modů (viz kap.5)
3. Účinek instrukcí na jednotlivé bity stavového registru (viz kap.3)

ADC Přičítání s přenosem

Příklad: ADC #2; přičti 2 k obsahu akumulátoru

Instrukce přičte k akumulátoru hodnotu argumentu a rovněž hodnotu přenosu, t.j. bit C ve stavovém registru.
Je-li výsledkem sčítání číslo menší než 256, bude po provedení instrukce bit C=0. Je-li výsledkem číslo >=256, nahodí se C=1 a v akumulátoru bude výsledek sčítání mínus 256.
Pokud pracuje 6502 v dekadickém módu, je největší číslo, které může být uloženo v akumulátoru 99 a přenos je generován již při překročení této hodnoty.
V závislosti na výsledku sčítání se mohou změnit i další bity stavového registru:

Bit překročení V se nahodí, pokud se sčítáním změní hodnota nejvýznamějšího bitu (bitu 7).

Bit N se nahodí, je-li výsledek sčítání >127, t.j. pokud bit 7=1.

A konečně bit Z se nahodí, je-li výsledkem sčítání 0.

Několik příkladů:

A	N	Z	C	V	instrukce	A	N	Z	C	V	poznámka
2	0	0	0	0	ADC #3	5	0	0	0	0	obyčejné sečtení
2	0	0	1	0	ADC #3	6	0	0	0	0	sčítání spříponou
2	0	0	0	0	ADC #254	0	0	1	1	0	C a Z nahozeny
2	0	0	0	0	ADC #253	255	1	0	0	1	N a V nahozeny
253	1	0	0	0	ADC #6	3	0	0	1	1	C a V nahozeny N shozen
125	0	0	1	0	ADC #2	128	1	0	0	1	N a V nahozeny C shozen

Je zřejmé, že testováním jednotlivých bitů můžeme snadno zjistit řadu informací o výsledku sčítání. Existující instrukce pro testování těchto bitů jsou velmi často využívány.

Adresové mody

Instrukce ADC využívá všech 8 adresních úvodů, distabovaných v kap.5 pro instrukci LDA. Jsou zde uvedeny společně s počtem bytů paměti a počtem cyklů, které potřebují

Modus	Instrukce	Cykl	Bytů	Význam
bezprostřední	ADC #2	2	2	A+ #2

absolutní	ADC $3420	4	3	A+ obsah buňky $3420
stránka 0	ADC $F6	3	2	A+ obsah buňky $F6
stránka 0,X	ADC $F6,X	4	2	A+ obsah buňky ($F6+X)
absolutní,X	ADC$3420,X	4	3	A+ obsah buňky ($3420+X)
absolutní,Y	ADC$3420,Y	4	3	A+ obsah buňky ($3420+Y)
index nepř.	ADC($F6,X)	6	2	A+ obsah buňky, jejíž adresa je na ($F6+X)
nepř. index	ADC($F6),Y	5	2	A+ obsah buňky, jejíž adresa je na ($F6)+Y

AND logický součin

Tato instrukce provede logický součin operandu a akumulátoru. V binárním vyjádření provede AND #$0E při obsahu akumulátoru #5 toto:

1.př.	Hex	Binárně	2př.	Hex	Binárně
akumulátor	$5	/00000101		$93	/10010011
operand	$0E	/00001110		$1D	/00011101
výsledek	$4	/00000100		$11	00010001

Jednička se objeví pouze v tom sloupci, kde je 1 v akumulátoru i v operandu. Tato instrukce se využívá poměrně často k odhalování určitých částí bytu, jednoduše uděláme logický součin s $0F.

Vliv na bity stavového registru.
Instrukce AND dosadí bit Z=1 je-li výsledek 0 a naopak. Rovněž dosadí bit N=1 je-li výsledek >127 a N=0, je-li <128
Příklady:

A	Z	N	Instrukce	=>	A	Z	N	Význam
5	0	0	AND #8	=>	0	1	0	Z=1, protože A=0
$FE	0	1	AND #$5F	=>	$5E	0	0	N=0, protože výsledek<127

Adresní mody jsou stejné jako pro LDA a ADC.

ASL Aritmetický posuv doleva

Instrukce ASL posune obsah operandu o 1 bit doleva. Nejvýznamnější bit (bit 7) se přesune do bitu C stavového registru a do nejvýznamnějšího bitu (bit 0) se dosadí 0.
Instrukce může být použita pro násobení 2, protože posuv doleva zdvojí hodnotu každého bitu. Je však nutno mít na paměti, že při operandu >128 dojde k posuvu do bitu C a tuto situaci je nutno správně ošetřit, aby výsledek byl podle očekávání.

Vliv na stavový registr.
Bit C bude obsahovat nejvyznamnější bit operandu. Bit N dosazen podle nejvýznamnějšího bitu výsledku (t.j. podle bitu 6 původního čísla).
Bit Z bude dosazen Z=1, pokud výsledek bude=0

Příklady:

A	N	C	Z	Instrukce =>	A	N	C	Z
128	1	0	0	ASL A	0	0	1	1
64	0	0	0	ASL A	128	1	0	0
192	1	0	0	ASL A	128	1	1	0
8	0	0	0	ASL A	16	0	0	0

Adresní mody, použitelné s instrukcí ASL následují

Modus	Instrukce	Cyklů	Bytů	význam
Akumulátor	ASL A	2	1	posuv bude v akumulátoru
Absolutní	ASL $3420	6	3	posuv bude v buňce $ 3420
Stránka 0	ASL $F6	5	2	posuv bude v buňce $ F6
Stránka 0,X	ASL $F6,X	6	2	posuv bude v buňce ($F6+X)
Absolutní,X	ASL$3420,X	7	3	posuv bude v buňce ($3420,X)

Tato instrukce trvá poněkud dlouho, ale pokud uvážíme, že nám umožní vynásobit číslo 2 za <4 mikrosekundy, je to dost rychlé, zejména ve srovnání s Basicem.

BCC Větvení při C=0

Tato instrukce způsobí změnu pořadí instrukcí za podmínky C=0. Pokud C=1, program pokračuje ve vykonávání dalších instrukcí v řadě. Instrukce má závažné omezení, instrukce, na kterou se skáče, musí být v rozmezí +-127 bytů programu, protože hodnota po větvení je obsažena v jednom bytu a číslo větší než 127 je bráno jako negativní a znamená skok zpět.

```
Příklad:    BCC  SKIP
            LDA  $0342 Větvení dopředu
      SKIP  LDA  $4582

Jiný příklad:
      LOOP  LDA $0342
            BCC LOOP  Větvení dozadu
```

Vliv na stavový registr: žádný
Adresní mody: jediný adresní modus je relativní
Větvení je dopředu nebo dozadu vůči okamžité hodnotě programového čítače, který ukazuje bezprostředně za instrukci BCC, t.j. na instrukci, která následuje.
BCC vyžaduje 2 byty v paměti a 2 cykly k provedení

BCS Větvení při C=1

Instrukce je přímým opakem instrukce BCC. Odskok se udělá ,pokud C=1 v opačném případě se pokračuje na následující instrukci.
Vliv na stavový registr: žádný
Adresní modus: pouze relativní
Potřebuje 2 byty v paměti a 2 cykly k provedení

BEQ Větvení při Z=1

Tato instrukce je obdobná k BCC a BCS, ale k rozhodnutí o větvení používá bit Z stavového registru. Bit Z=1 pokud výsledkem operace je 0 v akumulátoru připadá, jsou-li při porovnávání instrukce CMP, CPX, CPY oba porovnávací operandy stejné.
Vliv na stavový registr: žádný
Adresní modus: relativní
Potřebuje 2 byty paměti a 2 cykly k provedení

BIT Testový bit v paměti s akumulátorem

Tato instrukce provede logický součin operandu s akumulátorem, výsledek se však nezapíše do akumulátoru (jako obsah nemění). Výsledek se projeví pouze v bitech stavového registru.
Vliv na stavový registr:Změní se 3 bity stavového registru.Bit N je dosazen podle nejvýznamnějšího bitu (bitu 7) operandu. Bit V je dosazen podle hodnoty bitu 6 operandu. Bit Z je dosazen v závislosti na výsledku logickeho součinu operandu a akumulátoru. Je-li výsledek 0, je Z dosazeno Z=1, v opačném případě Z=0.
Příklady:Předpokládáme, že $0345 obsahuje hodnotu $F3.

A	N	V	Z	Instrukce	Výs.log.sou.	A	N	V	Z
128	1	0	0	BIT $0345	128	128	1	1	0
5	0	0	0	BIT $0345	1	5	1	1	0
4	0	0	0	BIT $0345	0	4	1	1	1
2	0	0	1	BIT $0345	3	3	1	1	0

Tato instrukce se používá především, chceme-li se něco dovědět o čísle v paměti.
Adresní mody:
Jsou k dispozici pouze 2, absolutní a stránka 0.

Modus	Instrukce	Cyklů	Bytů	Význam
absolutní	Bit $3420	4	3	viz výše
Stránka 0	Bit $F6	3	2	viz výše

BMI Větvení při N=1

Je to další instrukce podmíněného větvení k rozhodování používá bitu N stavového registru. Skok se vykoná při N=1, v opačném případě program pokračuje. Ve všech ostatních vlastnostech odpovídá instrukci BCC.
Vliv na stavový registr: žádný
Adresní modus: relativní
Vyžaduje 2 byty v paměti a 2 cykly na provedení

BNE Větvení při Z=0

Opak instrukce BEQ.
Vliv na stavový registr: žádný
Adresní modus: relativní
Potřebuje 2 byty v paměti a 2 cykly na provedení

BPL Větvení při kladném výsledku

Instrukce je přímým opakem BMI. K větvení dojde, pokud N=0 a nikoli, když N=1.
Vliv na stavový registr: žádný
Adresní modus: relativní
Potřebuje 2 byty v paměti a 2 cykly na provedení

BRK Instrukce BREAK

Hlavní využití této instrukce je při ladění programů. Využijete-li BRK ve svém programu a spustíte jej, většina dostupných ladicích monitorů (debugger) vypíše hodnoty registrů.

Vliv na stavový registr: žádný
Adresní modus: implikovaný
potřebuje 1 byte v paměti a 7 cyklů na provedení

BVC Větvení při V=0

Větvení v závislosti na bitu V, vše jako u BCC.
Vliv na stavový registr: žádný
Adresní modus: relativní
Potřebuje 2 byty v paměti a 2 cykly na provedení

BVS Větvení při V=1

Opak instrukce BVC, k větvení dojde při V=1
Vliv na stavový registr: žádný
Adresní modus: relativní
Potřebuje 2 byty v paměti a 2 cykly na provedení

CLC Shození bitu C stavového registru

Instrukce CLC dosadí bit C=0 bez ohledu na předchozí stav. Nejčastěji se používá ve spojení s instrukcí ADC. Protože instrukce ADC vždy přičte hodnotu C k součtu, je nutné zajistit, aby byla jeho hodnota před sčítáním rovna 0.
Typické použití pro sečtení 2 čísel bude.

```
LDA $6C
CLC       Sečti obsah buňky $6C
ADC #3    a 3, výsledek je v akumulátoru.
```

Vliv na stavový registr: změní hodnotu C na 0
Adresní modus: Implikovaný
Potřebuje 1 byte v paměti a 2 cykly na provedení

CLD Shození dekadického modu

Instrukce CLD dosadí ve stanoveném registru bit D=0 a tak převede posuv do binárního modu.Ve většině příkladů v této knize je užito binárního modu.V dekadickém modu se každá polovina bytu použije k zakódování jedné dekadické číslice podle tabulky.

Bin	číslice
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9

Rozdíl mezi dekadickými a binárním počítáním se ukáže při provádění aritmetických instrukcí.Např.

```
LDA $0345  ;obsahuje $59
CLC
ADC $0302  ;obsahuje $13
```

V binárním modu bude výsledek $6C.Pokud však bude posuv v dekadickém modu, výsledek bude $72, protože byty na $0345 a $0302 budou interpretovány jako dvojice dekadických číslic:59+13=72
Vliv na stavový registr: Dosadí bit D=0
Adresní modus: Implikovaný
Potřebuje 1 byte v paměti a 2 cykly na provedení.

CLI Shození bitu I (přerušení)

Instrukce pracuje přímo ve stavovém registru, bit I je nastaven na 0 a tím je povoleno maskovatelné přerušení. Pokud I=1, maskovatelná přerušení jsou zakázána. ATARI využívá mnoho typů přerušení a pokud I=1, nemůže k přerušení dojít. Do této kategorie přerušení patří i přerušení od vertikálního zatmívání a od display listu. Instrukce CLI tato přerušení povolí.
Vliv na stavový registr: Dosadí bit I=1
Adresní modus: Implikovaný
Potřebuje 1 byte v paměti a 2 cykly na provedení

CLV Shození bitu V (přetečení)

Instrukce dosadí V=0
Vliv na stavový registr: Dosadí v=0
Adresní modus: Implikovaný
Potřebuje 1 byte v paměti a 2 cykly na provedení

CMP Porovnej paměť s akumulátorem

Instrukce CMP nám umožní porovnat paměť s operandem. Při provádění této instrukce se od obsahu akumulátoru odečte operand a podle výsledku odečítání se dosadí hodnoty bitů ve stavovém registru. Obsah akumulátoru se nezmění. Podle změny bitů stavového registru zjišťujeme výsledek porovnání
Vliv na stavový registr: Instrukce CMP dosadí bit Z=1 v případě, že porovnaná čísla jsou si rovna. Jsou-li různá, dosadí se Z=0. Bit N se dosadí (N=1), pokud je výsledek odečítání větší než 127 (t.j nejzávažnější bit rozdílu je 1). Bit C (přetečení) je dosazen C=1, pokud operand je menší nebo roven hodnotě akumulátoru.
Příklady:(Předpokládáme, že buňka $0345 obsahuje 26).

A	Z	N	C	Instrukce	výs.odeč	Z	M	C	Poz.
26	0	0	0	CMP$0345	0	1	0	1	A= $0345
48	0	0	0	CMP$0345	22	0	0	1	A> $0345
130	0	1	0	CMP$0345	104	0	0	1	A- $0345<128
8	0	0	0	CMP$0345	238	0	1	0	A- $0345>127

Je důležité si uvědomit, že CMP nepoužívá znaménkovou aritmetiku, ale porovnává čísla mezi 0-255. Pokud používáte aritmetiky se znaménkem, je nutné to uvážit při interpretaci výsledků. Uvědomte si ještě, které instrukce větvení se provedou při daném obsahu akumulátoru a paměti. Předpokládáme výsledný program:

```
LDA #...
CMP $...
B...cíl
A Operand Odskočí      Neodskočí
8    8    BEQ,BCS,BPL BNE,BCC,BMI
9    8    BCS,BPL,BNE BCC,BMI,BEQ
8    9    BMI,BCC,BNE BPL,BCS,BEQ
```

Adresní mody: Tytéž jako při instrukci LDA,ADC

CPX Porovnej registr X a paměť

Tato instrukce porovnává hodnotu registru X s hodnotou operandu na rozdíl od CMP, kde se s pamětí porovnává obsah akumulátoru. Ve všech ostatních aspektech jsou instrukce identické. CPX se většinou používá k odstraňování hodnoty v registru X, zvláště je-li použit jako index, ke zjištění, zda již dosáhl požadovanou hodnotu.
Vliv na stavový registr:
Jsou-li čísla stejná, dosadí u Z=1, jinak Z=0. Je-li rozdíl X-operand větší jak 127, dosadí n=1, jinak N=0. Jestli-že číslo v paměti je <=X, dosadí se c=1,jinak c=0.
Adresní mody:

Módus	Instrukce	cyklů	bytů	výz.	
Bezprostřední	CPX#2	2	2	X-#2	
Absolutní	CPX$3420	4	3	X- obsah buňky	$3420
Stránka 0	CPX$F6	3	2	X- obsah buňky $F6	

CPY Porovnej registr Y a paměť

Platí totéž zo pro CPX nebo CPY, porovnává se registr Y s pamětí.
Vliv na stavový registr: Jako CPX
Adresní modus: Jako CPX

DEC Zmenši obsah paměti o 1

Instrukce se používá pro změnění obsahu paměti o 1. Teoreticky by bylo možno udělat totéž s akumulátorem a instrukcí odečítání, což by bylo mnohem složitější.
Vliv na stavový registr: Pokud výsledek odečtení 1 je 0,dosadí se Z=1, je-li výsledek <>0 dosadí se za Z=0. Je-li výsledek > 127, dosadí se N=1, jinak N=0.Je tedy možné zjistit výsledek operace, aniž by se číslo dávalo do akumulátoru, jednoduše testováním příslušných bitů ve stavovém registru.
Adresní mody:

Módus	Instrukce	Cuklů	Bitů	Význam
Absolutní	DEC$3420	6	3	zmenšení buňky $3420
Stránka 0	DEC$F6	5	2	změna buňky $F6
Stránka0,X	DEC$F6	6	2	změna buňke ($F6+X)
Absolutní,X	DEC$3420,X	7	3	změna buňky ($3420+X)

DEX Zmenši hodnotu v registru X o 1

Instrukce zmenšuje obsah registru X o 1 a používá se zvláště tehdy, když se registr X užívá jako index.
Vliv na stavový registr: Podobně jako DEC, tato instrukce nastaví bit Z a N.

Adresní modus: Implikovaný
Potřebuje 1 byte v paměti a 2 cykly na provedení

DEY Snížení hodnoty v registru Y o 1

Odpovídá instrukci DEX pro Y.
Vliv na stavový registr: Jako DEX
Adresní modus: Implikovaný
Potřebuje 1 byte v paměti a 2 cykly na provedení

EOR Exklusivní OR

Instrukce EOR v mnohém připomíná instrukci AND, na rozdíl od ní se bit po bitu provede instrukce exklusivní OR, t.j. ve výsledku je 1, pokud se liší příslušné bity operandů a 0 právě tehdy, když jsou si odpovídající bity rovny.
Příklad:

	Dek	Hex	binárně
Akumulátor	138	85	%10000101
Operand	186	BA	%10111010
Výsledek	63	3F	%00111111

Instrukce EOR může být využita k získávání doplňku bytů (********* všech 0 na 1 a naopak) tím, že se udělá EOR čísla s $FF.
Vliv na stavový registr:
Je-li výsledek >127, dosadí se N=1, jinak N=0
Je-li výsledek 0, dosadí se za Z = 1, jinak Z = 0
Adresní mody: stejné jako u instrukcí LDA, ADC

INC zvětšit hodnotu paměti o 1

Instrukce je opak instrukce DEC, způsobí zvětšení hodnoty v buňce paměti o 1.
Vliv na stavový registr:
dosadí N=1, pokud je výsledek>127 jinak N=0
rovněž se dosadí Z=1, je-li výsledek 0, jinak Z=0.
Adresní mody:tytéž jako DEC

INX Zvětši hodnotu registru X o 1

Instrukce je opakem instrukce DEX, zvětší hodnotu v registru X o 1.
Vliv na stavový registr: Jako DEX
Adresní mody: Jako DEX

INY Zvětši hodnotu registru Y o 1

Podobně jako INX je INY opakem DEY zvětší hodnotu v registru Y o 1.
Vliv na stavový registr: Jako DEX
Adresní modus: Jako DEX

JMP Skok na adresu

Instrukce provede skok na adresu daného operandu bez ohledu na jaké koli podmínky.
Vliv na stavový registr: žádný
Adresní mody: jsou dva, absolutní a nepřímý. V absolutním modu za kódem instrukce následují přímo dva byty adresy, na niž se předá řízení. Například: JMP$ 0423 (potřebuje 3 byty, 3 cykly). V nepřímém adresním modu se na adrese, udání v operandu najde vlastní adresa, na kterou se má skočit. Vyznačí se uzavřením operandu do závory: JMP ($0432) nebo JMP (0)
Jestliže na adrese $0423,$0424 jsou uloženy byte $26,$06, provede se skok na adresu $0626. V nepřímém modu jsou zapotřebí 3byty a 5cyklů na provedení.

JSR Skok do podprogramu

Instrukce provede skok na adresu, udanou v operandu a zatím uloží do zásobníku ve str.1 hodnotu čítače instrukcí t.j. adresu instrukcí, která následuje za instrukcí JSR, což později umožní návrat z podprogramu instrukcí RST.
Vliv na stavový registr: žádný
Adresní modus: pouze absolutní
3 byty v paměti, 6 cyklů

LDA Naplň akumulátor

Instrukce naplň akumulátor hodnotou operandu. Společně s instrukcí STA je to pravděpodobně nejčastěji užívaná instrukce 6502.
Vliv na stavový registr: Je-li číslo naplněné do akumulátoru >127 dosadí se N=1, jinak N=0. Je-li do registru naplněná 0 dosadí se Z=1, jinak Z=0
Adresní modus: Tutéž jako pro LDA

LDX Naplň registr X

Instrukce naplň registr X hodnotou operandu.
Vliv na stavový registr: Je-li naplněné číslo >127, dosadí se N=1, jinak N=0. Je-li naplněna 0 dosadí se Z=1, jinak Z=0

Adresní mody:

Módus	Instrukce	Cyklů	Bytů
Bezprostřední	LDX#2	2	2
Absolutní	LDX$3420	4	3
Stránka 0	LDX$F6	3	2
Stránka 0,Y	LDX$F6,Y	4	2
Absolutní,Y	LDX$3420,Y	4	3

LDY Naplň registr Y

Naplň registr Y hodnotou operandu
Vliv na stavový registr: Jako LDX
Adresní mody:

Módus	Instrukce	Cyklů	Bytů
Bezprostřední	LDY#2	2	2

Absolutní	LDY$3420	4	3
Stránka 0	LDY$F6	3	2
Stránka 0,x	LDY$F6,X	4	2
Absolutní,X	LDY$3420,X	4	3

LSR Posuv do prava

Instrukce je s opačným významem k ASL. Instrukce dosadí 0 do nejvýznamnějšího bitu (bitu 7) a posune každý bit o 1 pozici níže. Bit 0 se přitom přesune do do bitu C ve stanovém registru.

Bity	7654321	C (přenos)
před	10110101	0
Po	01011010	1

Instrukce LSR efektivně způsobí vydělení 2, což však neplatí u znaménkové aritmetiky (do bitu 7 se vždy dosadí 0 bez ohledu na původní obsah).
Vliv na stavový registr: Bit N se vždy dosadí N=0. Bit Z se dosadí Z=1, je-li výsledkem 0, jinak Z=0. Bit C (přenos) se naplní původním obsahem bitu 0.
Adresní mody:

Módus	Instrukce	Cyklů	Bytů
Absolutní	LSR$3420		
Stránka 0			
Stránka 0,X			
Absolutní,X			
Akumulátor			

NOP Prázdná operace

Instrukce nekoná žádnou činnost. Můžeme jí využít pro realizaci místa v programu nebo pro připsání zbytečné instrukce.
Vliv na stavový registr: žádný
Adresní modus: Implikovaný
Počet bytů v paměti 1, 2 cykly

ORA Logický součet paměti a akumulátoru

Je to poslední ze 3 logických instrukcí AND a EOR. Porovnají se jednotlivé bity akumulátoru a operandu a výsledek se uloží do akumulátoru. Je-li alespoň jeden z odpovídajících bitů 1, je výsledkem 1, jsou-li oba nulové, je výsledkem 0.
Příklad:

1 číslo	%10100101
2 číslo	%01101100
výsledek	%11101101

Hlavní účel ORA je dosadit konkrétní bit do 1.
Vliv na stavový registr: Je-li operace=0, pak Z=1, jinak Z=0. Je-li výsledněn číslo>127, pak N=1, jinak N=0
Adresní mody: stejné jako LDA, ADC

PHA Ulož akumulátor do zásobníku

K přechodnému uložení informace můžeme použít více míst. Může to být rezervovaná buňka v paměti, některý z registrů ap. Jediná metoda, která nepoškodí žádnou jinou informaci (jak by mohly instrukce TAX nebo TAY) je použít PHA, která uloží obsah akumulátoru do zásobníku a zmenší hodnotu ukazatele zásobníku o 1. Je však nutno mít na paměti, že zásobník je použit na cílové návratové adresy z podprogramu a že musíme před instrukcí RST uvést zásobník do původního stavu. Dalším omezením je velikost zásobníku, operací rozsahem 1 stránky.
Vliv na stavový registr: žádný
Adresní modus: Implikovaný

1byte, 3 cykly

PHP Ulož stavový registr do zásobníku

Tato instrukce uloží stavový registr do zásobníku. Jejím účelem je zachovat stavové bity pro pozdější využití. Podobně jako u instrukce PHA je nutno dát pozor, aby informace v zásobníku byly vybírány ve správném pořadí a nedošlo ke kolizi s informacemi instrukcí JSR.
Vliv na stavový registr - žádný
Adresní modus: Implikovaný
1 byte, 3 cykly

PLA Naplň akumulátor ze zásobníku

Je protějškem instrukce PHA. Vyjme byte, který je na vršku zásobníku a uloží ho do akumulátoru, upraví při tom hodnotu ukazatele zásobníku.

Při použití strojových podprogramů pro Basic je instrukce PLA důležitá při přebírání argumentů z volání USR. Každý podprogram volaný z Basicu, musí začínat instrukcí PLA, kterou do akumulátoru přeneseme počet parametrů při volání CALL. <Blíže viz. kap.7>

Vliv na stavový registr: Je-li hodnota, přenesená ze zásobníku do akumulátoru > 127, dosadí se N=1, jinak N=0. Je-li tato hodnota rovna 0, dosadí se Z=1 a naopak.
Adresní modus: Implikovaný
1 byte, 4 cykly

PLP Naplň stavový registr ze zásobníku

PLP je inverzní k PHP a přesouvá informaci ze zásobníku do stavového registru. Obvykle slouží k obnovení stavu, který byl v okamžiku, kdy se provedla instrukce PHP.
Vliv na stavový registr: Instrukce změní všechny bity stavového registru tím, že registr naplní hodnotami z vršku zásobníku.
Adresní modus: Implikovaný
1 byte, 4 cykly

RTS Návrat z podprogramu

Instrukce RTS vrací řízení na instrukci, která následuje ve volajícím programu za instrukcí JSR. Programový čítač je dosazen podle hodnoty, která je v tomto okamžiku na vrcholu zásobníku. Je proto důležité, aby stav zásobníku odpovídal okamžiku, kdy bylo prováděno JSR.

Vliv na stavový registr:Žádný
Adresní modus:Implikovaný
1 byte, 6 cyklů

RTI Návrat z podprogramu přerušení

Istrukce RTI vrací řízení programu těsně za místo, kde došlo k přerušení.Při zpracování přerušení se do zásobníku ukládá nejenom návratová adresa, ale i obsah stavového registru.
Vliv na stavový registr:nastaví se taková hodnota bitů, jaká byla před zpracováním přerušení.
Adresní modus:Implikovaný
1 byte, 6 cyklů

SBC Odečítání s přenosem

Je to jediná instrukce sloužící k odečítání. Od hodnoty akumulátoru se odečte hodnota operandu a výsledek se uloží do akumulátoru.Pokud je obsah bitu C=1, je výsledek o 1 menší. Proto je nutno zajistit před odečítáním, aby bit C=1, např. instrukcí SEC. Bitu C můžeme využít při vícebytové aritmetice, kde postupně odečítáme byty od nejméně významných a přenos v bitu C nám zajistí správný výsledek.
Vliv na stavový registr:Bit C se dosadí v závislosti na vzájemné velikosti hodnoty v akumulátoru a operandu, pokud byla hodnota operandu menší, dosadí se C=1, jinak C=0.Pokud se výsledek rovná nule, dosadí se Z=1, jinak Z=0. Je-li výsledek > než 127, dosadí se V=1, jinak V=0.
Adresní mody: Tytéž jako pro LDA, ADC

SEC Dosaď bit C=1

Tuto instrukci použijeme vždy, když potřebujeme bit C=1.Hlavní použití je při odečítání před instrukcí SBC. Jiné použití je před instrukcí rotace
Vliv na stavový registr:Dosadí C=1
Adresní modus:Implikovaný
1 byte, 2 cykly

SED Dosaď dekadický modus

6502 může pracovat v binárním nebo dekadickém modu.K přechodu do binárního modu se používá instrukce CLD, do dekadického instrukce SED.Po instrukci SED budou všechna sčítání a odečítání v dekadickém modu, pokud nebude do bitu D dosazena nula, např instrukcí CLD.
Vliv na stavový registr:Dosadí D=1
Adresní modus:Implikovaný
1 byte , 2 cykly

SEI Dosaď bit I<zákaz přerušení>

Dosazení I=1 znemožní maskovatelné
přerušení jako přerušení od display-
listu nebo od svislého zatemňovacího impulsu. Za určitých okolností potřebujeme vlastní přerušovací podprogram.V takovém případě je vhodné

dosadit bit I=1,sdělit 6502 adresu našeho programu pro obsluhu přerušení dosazením adresy v příslušném vektoru <viz Kap.8> a pak dosadit I=0,pro normální pokračování.Tím znemožníme přerušení ,aby se projevilo v okamžiku ,kdy měníme adresu ve vektoru přerušení.Kdyby k takovému přerušení došlo v polovině práce,vedlo by to bezpochyby k zhroucení programu.
Vliv na stavový vektor: dosadí se I=1
Adresní modus :Implikovaný
1 byte ,2 cykly

STA Ulož obsah akumulátoru do paměti

Instrukce STA uloží obsah akumulátoru kamkoliv do paměti.Obsah akumulátoru se přesunem nezmění .
Vliv na stavový registr :žádný
Adresní mody: Využívá 7 z 8 adresních mod0 LDA.

MODUS	INSTRUKCE	CYKLY	BYTY
ABSOLUTNÍ	STA$3420	4	3
STRÁNKA 0	STA$F6	3	2
STRÁNKA 0,X	STA$F6,X	4	2
ABSOLUTNÍ,X	STA$3420,X	4	3
ABSOLUTNÍ,Y	STA$3420,Y	4	3
INDEX.NEPŘ.	STA(&F6,X)	6	2
NEPŘ.INDEX.	STA(&F6),Y	5	2

STX Ulož obsah registru X do paměti.

Instrukce uloží do paměti obsah registru X.
Vliv na stavový registr :žádný
Adresní mody:

MODUS	INSTRUKCE	CYKLY	BYTY
ABSOLUTNÍ	STX$3420	4	3
STRÁNKA 0	STX$F6	3	2
STRÁNKA 0,Y	STX$F6,Y	4	2

STY Ulož obsah registru Y do paměti.

Instrukce uloží obsah registru Y do paměti.
Vliv nastavový registr : žádný
Adresní mody:

MODUS	INSTRUKCE	CYKLY	BYTY
ABSOLUTNÍ	STY$3420	4	3
STRÁNKA 0	STY$F6	3	2
STRÁNKA 0,X	STY$F6,X	4	2

TAX Přenos obsahu akumulátoru do registru X

Instrukce přenese obsah akumulátoru do registru X,akumulátor se nezmění.

Vliv na stavový registr :Je-li přenášené číslo >127, dosadí se N=1,jinakN=0.Je-li přenášené číslo =0,dosadí se Z=1,jinakZ=0.
Adresní modus:Implikovaný
1 byte, 2 cykly

TAY Přenos obsahu akumulátoru do registru Y.

Instrukce přenese obsah akumulátoru do registru Y.
Vliv na stavový registr:Je-li číslo v akumulátoru >127 ,dosadí se N=1,jinak N=0.Je-li toto číslo =0,dosadí se Z=1 ,jinak Z=0.
Adresní modus: Implikovaný
1 byte, 2 cykly

TSX Přenes ukazatel zásobníku do registru X

Instrukce přenese hodnotu ukazatele zásobníku do registru X,obvykle před jeho uchováním pro další použití.
Vliv na stavový registr:Je-li hodnota ukazatele>127 <což obvykle je>,dosadí se N=1,jinak N=0.Byla-li hodnota ukazatele =0<téměř nikdy>,dosadí se Z=1,jinak Z=0.
Adresní mod: Implikovaný
1 byte , 2 cykly

TXA Přenes obsah X-registru do akumulátoru

Instrukce přenese hodnotu z X-registru do akumulátoru beze změny hodnoty v X.
Vliv na stavový registr:Je-li přenášené číslo >127 ,dosadí se N=1,jinak N=0.Bylo-li číslo =0 dosadí se Z=1,jinak Z=0.
Adresní modus: Implikovaný
1 byte, 2 cykly

TXS Přenes obsah registru Xdo ukazatele zásobníku

Tato instrukce se často používá pro nastavení ukazatele zásobníku na určitou hodnotu.TXS je nutno používat s maximální opatrností.Ničím se nedá obsah zásobníku pokazit víc než touto instrukcí.
Vliv na stavový registr: žádný
Adresní modus: Implikovaný
1 byte, 2 cykly

TYA Přenes obsah registru Y do akumulátoru

Instrukce přenese obsah registru Y do akumulátoru .
Vliv na stavový registr :Je-li přenášené číslo >127,pak se dosadí N=1,jinak N=0.
Adresní modus:Implikovaný
1 byte , 2 cykly

PRILOHA 2

DEC.	HEX.	ATASCII	INTERN	KEY COMB.	KEY CODE
0	$0	♥		CTRL ,	L
1	$1	├	!	CTRL A	J
2	$2	│	"	CTRL B	;
3	$3	┘	#	CTRL C	F1 1200XL
4	$4	┤	$	CTRL D	F2 1200XL
5	$5	┐	%	CTRL E	K
6	$6	╱	&	CTRL F	+
7	$7	╲	'	CTRL G	*
8	$8	◢	(	CTRL H	O
9	$9	▗	)	CTRL I	
10	$A	▶	*	CTRL J	P
11	$B	▖	+	CTRL K	U
12	$C	▝	,	CTRL L	RETURN
13	$D	▔	-	CTRL M	I
14	$E	▁	.	CTRL N	-
15	$F	▘	/	CTRL O	=
16	$10	♣	0	CTRL P	V
17	$11	▛	1	CTRL Q	HELP
18	$12	▀	2	CTRL R	C
19	$13	▜	3	CTRL S	F3 1200XL
20	$14	●	4	CTRL T	F4 1200XL
21	$15	▟	5	CTRL U	B
22	$16	▌	6	CTRL V	X
23	$17	┬	7	CTRL W	Z
24	$18	┴	8	CTRL X	4
25	$19	▋	9	CTRL Y	
26	$1A	└	:	CTRL Z	3
27	$1B	↰	;	↰/↰	6
28	$1C	↑	<	↰ CTRL -	ESCAPE
29	$1D	↓	=	↰ CTRL =	5
30	$1E	←	>	↰ CTRL +	2
31	$1F	→	?	↰ CTRL *	1
32	$20		@	SPACE	,
33	$21	!	A	SHIFT 1	SPACE
34	$22	"	B	SHIFT 2	.
35	$23	#	C	SHIFT 3	N
36	$24	$	D	SHIFT 4	
37	$25	%	E	SHIFT 5	M
38	$26	&	F	SHIFT 6	/
39	$27	'	G	SHIFT 7	INVERSE
40	$28	(	H	SHIFT 9	R
41	$29	)	I	SHIFT 0	
42	$2A	*	J	*	E
43	$2B	+	K	+	Y
44	$2C	,	L	,	TAB
45	$2D	-	M	-	T
46	$2E	.	N	.	W
47	$2F	/	O	/	Q
48	$30	0	P	0	9
49	$31	1	Q	1	
50	$32	2	R	2	0

DEC.	HEX.	ATASCII	INTERN	KEY COMB.	KEY CODE
51	$33	3	S	3	7
52	$34	4	T	4	DEL/B/SP
53	$35	5	U	5	8
54	$36	6	V	6	<
55	$37	7	W	7	>
56	$38	8	X	8	F
57	$39	9	Y	9	H
58	$3A	:	Z	SHIFT ;	D
59	$3B	;	[	;	
60	$3C	<	\	<	CAPS
61	$3D	=	]	=	G
62	$3E	>	^	>	S
63	$3F	?	-	SHIFT /	A
64	$40	@	♥	SHIFT 8	SHIFT L
65	$41	A	▌	A	SHIFT J
66	$42	B	▐	B	SHIFT ;
67	$43	C	▄	C	SHIFT F1
68	$44	D	▖	D	SHIFT F2
69	$45	E	▗	E	SHIFT K
70	$46	F	▝	F	SHIFT +
71	$47	G	▘	G	SHIFT *
72	$48	H	▙	H	SHIFT O
73	$49	I	▚	I	
74	$4A	J	▛	J	SHIFT P
75	$4B	K	▜	K	SHIFT U
76	$4C	L	▝	L	SHIFT RETURN
77	$4D	M	▔	M	SHIFT I
78	$4E	N	▁	N	SHIFT -
79	$4F	O	▒	O	SHIFT =
80	$50	P	♣	P	SHIFT V
81	$51	Q	▗	Q	SHIFT HELP
82	$52	R	▔	R	SHIFT C
83	$53	S	▖	S	SHIFT F3
84	$54	T	●	T	SHIFT F4
85	$55	U	▂	U	SHIFT B
86	$56	V	▕	V	SHIFT X
87	$57	W	▘	W	SHIFT Z
88	$58	X	▗	X	SHIFT 4
89	$59	Y	▐	Y	
90	$5A	Z	▖	Z	SHIFT 3
91	$5B	[	♀	SHIFT ,	SHIFT 6
92	$5C	\	↑	SHIFT +	SHIFT ESCAPE
93	$5D	]	↓	SHIFT .	SHIFT 5
94	$5E	^	←	SHIFT *	SHIFT 2
95	$5F	-	→	SHIFT -	SHIFT 1
96	$60	♦	♦	CTRL .	SHIFT ,
97	$61	a	a	A	SHIFT SPACE
98	$62	b	b	B	SHIFT .
99	$63	c	c	C	SHIFT N
100	$64	d	d	D	
101	$65	e	e	E	SHIFT M
102	$66	f	f	F	SHIFT /
103	$67	g	g	G	SHIFT INVERS
104	$68	h	h	H	SHIFT R
105	$69	i	i	I	

DEC.	HEX.	ATASCII	INTERN	KEY COMB.	KEY CODE
106	$6A	j	j	J	SHIFT E
107	$6B	k	k	K	SHIFT Y
108	$6C	l	l	L	SHIFT TAB
109	$6D	m	m	M	SHIFT T
110	$6E	n	n	N	SHIFT W
111	$6F	o	o	O	SHIFT Q
112	$70	p	p	P	SHIFT 9
113	$71	q	q	Q	
114	$72	r	r	R	SHIFT 0
115	$73	s	s	S	SHIFT 7
116	$74	t	t	T	SHIFT D.B.SP.
117	$75	u	u	U	SHIFT 8
118	$76	v	v	V	SHIFT <
119	$77	w	w	W	SHIFT >
120	$78	x	x	X	SHIFT F
121	$79	y	y	Y	SHIFT H
122	$7A	z	z	Z	SHIFT D
123	$7B	♠	♠	CTRL ;	
124	$7C	|	|	SHIFT =	SHIFT CAPS
125	$7D	↑	↑	↑ SHIFT <	SHIFT G
126	$7E	←	←	↑ D.B.SP.	SHIFT S
127	$7F	▶	▶	↑ TAB	SHIFT A
128	$80			CTRL ,	CTRL L
129	$81			CTRL A	CTRL J
130	$82			CTRL B	CTRL ;
131	$83			CTRL C	CTRL F1
132	$84			CTRL D	CTRL F2
133	$85			CTRL E	CTRL K
134	$86			CTRL F	CTRL +
135	$87			CTRL G	CTRL *
136	$88			CTRL H	CTRL O
137	$89			CTRL I	
138	$8A			CTRL J	CTRL P
139	$8B			CTRL K	CTRL U
140	$8C			CTRL L	CTRL RETURN
141	$8D			CTRL M	CTRL I
142	$8E			CTRL N	CTRL -
143	$8F			CTRL O	CTRL =
144	$90			CTRL P	CTRL V
145	$91			CTRL Q	CTRL HELP
146	$92			CTRL R	CTRL C
147	$93			CTRL S	CTRL F3
148	$94			CTRL T	CTRL F4
149	$95			CTRL U	CTRL B
150	$96			CTRL V	CTRL X
151	$97			CTRL W	CTRL Z
152	$98			CTRL X	CTRL 4
153	$99			CTRL Y	
154	$9A			CTRL Z	CTRL 3
155	$9B	↵		RETURN	CTRL 6
156	$9C			↑ SH/D.B.S	CTRL ↑
157	$9D			↑ SHIFT >	CTRL 5
158	$9E			↑ CTRL TAB	CTRL 2
159	$9F			↑ SH/TAB	CTRL 1

DEC.	HEX.	ATASCII	INTERN	KEY COMB.	KEY CODE
160	$A0	■	0	SPACE	CTRL ,
161	$A1	▙	1	SHIFT 1	CTRL SPACE
162	$A2	▛	2	SHIFT 2	CTRL .
163	$A3	▀	3	SHIFT 3	CTRL N
164	$A4	▜	4	SHIFT 4	
165	$A5	▚	5	SHIFT 5	CTRL M
166	$A6	▞	6	SHIFT 6	CTRL /
167	$A7	▟	7	SHIFT 7	CTRL INVERS
168	$A8	▌	8	SHIFT 9	CTRL R
169	$A9	▐	9	SHIFT 0	
170	$AA	▄	:	:	CTRL E
171	$AB	▖	;	;	CTRL Y
172	$AC	▗	,	,	CTRL TAB
173	$AD	▔	-	-	CTRL T
174	$AE	▝	.	.	CTRL W
175	$AF	▘	/	/	CTRL Q
176	$B0	0	0	0	CTRL 9
177	$B1	1	1	1	
178	$B2	2	2	2	CTRL 0
179	$B3	3	3	3	CTRL 7
180	$B4	4	4	4	CTRL D.B.SP.
181	$B5	5	5	5	CTRL 8
182	$B6	6	6	6	CTRL <
183	$B7	7	7	7	CTRL >
184	$B8	8	8	8	CTRL F
185	$B9	9	9	9	CTRL H
186	$BA	+	+	SHIFT ;	CTRL D
187	$BB	*	'	*	
188	$BC	<	,	<	CTRL CAPS
189	$BD	=	-	=	CTRL G
190	$BE	>	.	>	CTRL S
191	$BF	?	/	SHIFT /	CTRL A
192	$C0	@	@	SHIFT 8	
193	$C1	A	A	A	
194	$C2	B	B	B	
195	$C3	C	C	C	
196	$C4	D	D	D	
197	$C5	E	E	E	
198	$C6	F	F	F	
199	$C7	G	G	G	
200	$C8	H	H	H	SH/CTRL O
201	$C9	I	I	I	
202	$CA	J	J	J	SH/CTRL P
203	$CB	K	K	K	SH/CTRL U
204	$CC	L	L	L	SH/CTRL RET
205	$CD	M	M	M	SH/CTRL I
206	$CE	N	N	N	SH/CTRL -
207	$CF	O	O	O	SH/CTRL =
208	$D0	P	P	P	
209	$D1	Q	Q	Q	
210	$D2	R	R	R	
211	$D3	S	S	S	
212	$D4	T	T	T	
213	$D5	U	U	U	
214	$D6	V	V	V	
215	$D7	W	W	W	

DEC.	HEX.	ATASCII	INTERN	KEY COMB.	KEY CODE
216	$D8	[illegible]	[illegible]	[illegible]	SH/CTRL 4
217	$D9	[illegible]	[illegible]	[illegible]	
218	$DA	[illegible]	[illegible]	[illegible]	SH/CTRL 3
219	$DB	[illegible]	[illegible]	[illegible]	SH/CTRL 6
220	$DC	[illegible]	[illegible]	[illegible]	SH/CTRL [illegible]
221	$DD	[illegible]	[illegible]	[illegible]	SH/CTRL 5
222	$DE	[illegible]	[illegible]	[illegible]	SH/CTRL 2
223	$DF	[illegible]	[illegible]	[illegible]	SH/CTRL 1
224	$E0	[illegible]	[illegible]	[illegible]	SH/CTRL ,
225	$E1	[illegible]	[illegible]	[illegible]	SH/CTRL SPAC
226	$E2	[illegible]	[illegible]	[illegible]	SH/CTRL .
227	$E3	[illegible]	[illegible]	[illegible]	SH/CTRL N
228	$E4	[illegible]	[illegible]	[illegible]	
229	$E5	[illegible]	[illegible]	[illegible]	SH/CTRL M
230	$E6	[illegible]	[illegible]	[illegible]	SH/CTRL /
231	$E7	[illegible]	[illegible]	[illegible]	SH/CTRL INVERS
232	$E8	[illegible]	[illegible]	[illegible]	SH/CTRL R
233	$E9	[illegible]	[illegible]	[illegible]	
234	$EA	[illegible]	[illegible]	[illegible]	SH/CTRL E
235	$EB	[illegible]	[illegible]	[illegible]	SH/CTRL Y
236	$EC	[illegible]	[illegible]	[illegible]	SH/CTRL TAB
237	$ED	[illegible]	[illegible]	[illegible]	SH/CTRL T
238	$EE	[illegible]	[illegible]	[illegible]	SH/CTRL W
239	$EF	[illegible]	[illegible]	[illegible]	SH/CTRL Q
240	$F0	[illegible]	[illegible]	[illegible]	SH/CTRL 9
241	$F1	[illegible]	[illegible]	[illegible]	
242	$F2	[illegible]	[illegible]	[illegible]	SH/CTRL 0
243	$F3	[illegible]	[illegible]	[illegible]	SH/CTRL 7
244	$F4	[illegible]	[illegible]	[illegible]	SH/CTRL D.B.SP
245	$F5	[illegible]	[illegible]	[illegible]	SH/CTRL 8
246	$F6	[illegible]	[illegible]	[illegible]	SH/CTRL <
247	$F7	[illegible]	[illegible]	[illegible]	SH/CTRL >
248	$F8	[illegible]	[illegible]	[illegible]	SH/CTRL F
249	$F9	[illegible]	[illegible]	[illegible]	SH/CTRL H
250	$FA	[illegible]	[illegible]	[illegible]	SH/CTRL D
251	$FB	[illegible]	[illegible]	[illegible]	
252	$FC	[illegible]	[illegible]	[illegible]	SH/CTRL CAPS
253	$FD	[illegible]	[illegible]	[illegible] CTRL 2	SH/CTRL G
254	$FE	[illegible]	[illegible]	[illegible] CT DBS	SH/CTRL S
255	$FF	[illegible]	[illegible]	[illegible] CTRL >	SH/CTRL A

Obsah:

Igi/2019