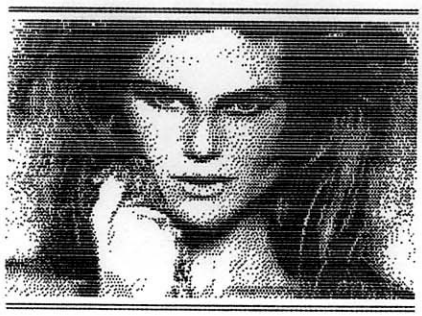
PROGRAM BITEN 90-5



ISSN 0281-1146

REDAKTÖREN

Det har inte kommit in något nytt från medlemmarna till detta nummer. Det finns dock tillräckligt mycket på lager från utlandet till årgång 1991 för att hjälpligt fylla fyra nummer med 24 sidor och två nummer med 15 sidor. I övrigt finns endast Tigercub att tillgå för 1992. Jag tror inte att föreningen kan leva kvar 1992 om inte medlemmarna börjar skriva artiklar för tidningen. Det behövs minst 4 medlemmar som regelbundet sänder in 4 sidor/nummer.

Funnelweb 4.30 finns nu med en förbättrad DR80 (för 9938) från SEP/03 /90. Den klarar av att visa Myartbilder med View File även om bildhuvudet har underligt innehåll vilket inte Barry Boones Myartladdare klarar av.

Föreningen har nu tre Geneve-ägare: Sören Bernle, Thomas Kolk och Leif Pedersen. Thomas Kolk efterlyser en beskrivning av vilka program som inte fungerar med en Geneve och om någon kan patcha SXB för Geneve. Det finns även två ägare till DIJIT AVPC: Jan Alexandersson och Bengt Jönsson. Mikael Nordlin har ett Mechatronic 80-kolumnskort. Det finns således 6 kända ägare av videoprocessorn 9938.

Det finns fyra kända ägare av Myarc HFDC med hårddisk: Bengt Jönsson, Jan Alexandersson, Leif Pedersen och John Hanssen (som övertagit kortet från Bertil Stenfeldt).

Sören Bernle har utökat sitt Myarc FDC med en 1,44 Mbytes skivenhet som används med 720 kbytes 3,5".

Manusstopp 15 nov för PB 90-6
15 jan för PB 91-1. ■

MEDLEMSMÖTE 17 NOV I STAFFANSTORP

Du inbjudes till nordiskt medlemsmöte lördagen 17 november 1990 kl. 10.00 i Staffanstorp. Tag kontakt med Sven Lundgren tel. 046-25 02 14 för att få en vägbeskrivning till möteslokalen. Danska 99/4A- och Geneve-ägare är även inbjudna och beräknas deltaga. ■

Redaktör: Jan Alexandersson
Utmanarredaktör: Anders Persson
TI-74 redaktör: Lars Herold Andersen
Programbankir: Börje Häll

Föreningens adress:
Föreningen Programbiten
c/o Schibler
Wahlbergsgatan 9 NB
S-121 46 JOHANNESHOV
Sverige

Postgiro 19 83 00-6
Medlemsavgiften för 1990 är 120:-

Datainspektionens licensnummer:
82100488

Annonser, insatta av enskild medlem (ej företag), som gäller försäljning av moduler eller andra tillbehör i enstaka exemplar är gratis.

Övriga annonser kostar 200 kr för hel sida. För lösblad (kopieras av annonsören) som skickas med tidningen gäller 200 kr per blad. Föreningen förbehåller sig rätten att avböja annonser som ej hör ihop med föreningens verksamhet eller ej på ett seriöst sätt gäller försäljning av originalexemplar av program.

Föreningens tillbehörsförsäljning:
Följande tillbehör finns att köpa genom att motsvarande belopp insätts på postgiro 19 83 00-6 (porto ingår)

Användartips med Mini Memory	20:-
Nittinian T-tröja	40:-
99er mag. 12/82, 1-5,7-9/83(st)	40:-
Nittinian årgång 1983	50:-
Programbiten årgång 84-89(styck)	50:-
TI-Forth manual	100:-
Hel diskett ur programbanken(st)	30:-

Enstaka program 5:- st + startkostnad 15 kr per skiva eller kassett (1 program=20kr, 3 program=30 kr). Se listor i PB89-3 och PB90-4.

Jan Alexandersson, Springarvägen 5, 3 tr, 142 61 TRÅNGSUND (08-7710569)

BASIC & XB TIPS(HTA) - 2

from Bill Sponchia, Canada

10. The manual tells you that there are 16 different character sets that you can redefine and change colors on. Actually there are 17 - Set #0 is never mentioned.

11. When LISTing a program and you see a line reference to "32767" this means (unless you actually used that line #) that you resequenced the program while you had a GOTO (or GOSUB, etc) to a non-existing line.

12. To LIST a portion of a program to the printer then enter the following command:
LIST "filename":line number range
eg: LIST "PIO":130-240 - will LIST to PIO lines 130 to 240 (inclusive)

13. If you have the a Ramdisk and are using VerMenu then you can go from your running BASIC (or XB) program to the Menu screen by putting in the following statement inplace of "END": DELETE "MENU"

14. Are you the type that likes to put use the computer but are always worried about accidently pressing the "QUIT" key. Here is a one line program that you can set up and run each time you sit down to do some work:

```
10 PRINT "QUIT KEY DISABLED"::CALL
   INIT::CALL LOAD(-31806,16)
```

If you save it as a program called LOAD then each time you start working then the program will automatically load and run thus taking care of all you worries (or at least some of them.)

15. Did you know that you could identify your GOSUB routines within the program without using the "!" or REM statement. You are allowed to put one word (string) after the GOSUB line number.
Here's an example program:

```
10 CALL CLEAR::PRINT "HERE I GO.."
20 GOSUB 50 DELAY_ROUTINE ::PRINT
   "I'M BACK!"
30 END
50 FOR T=1 TO 400::NEXT T::RETURN
```

16. Here are some interesting redefinitions for characters. To use them the proper format is "CALL CHAR (##,string) where "##" stands for the number of Character to be redefined and "string" is one of the following (or any other that you may have).

```
000804027F020408 = right arrow
00102040FE402010 = left arrow
081C2A4908080800 = up arrow
0008080B492AC108 = down arrow
00FF             = solid line
0000FE2828282828 = pi symbol
00083C4848483C08 = cent sign
0002020404482810 = check mark
```

17. The IMAGE statement
(eg - 100 IMAGE ###.##) can be used with the DISPLAY AT statement using the following format -

```
DISPLAY AT(5,12):USING 100:A
```

18. Instead of using the IMAGE statement you can define a variable in the image you would like the output to look and then say "USING variable name". eg -

```
100 F$="###.##"
110 DISPLAY AT(12,1):USING F$:A
```

Of course, unlike the IMAGE statement which can be anywhere in the program, the variable would have to be defined BEFORE using it in a DISPLAY AT or PRINT statement.

19. When using the DISPLAY AT statement you can use TAB to properly locate where further information is to be displayed.
eg: To set up the following display

```
MAIN MENU:
1 - Edit
2 - Add
3 - Exit
```

you can set up each line with an individual DISPLAY AT statement or you can do the following:

```
DISPLAY AT(5,5):"MAIN MENU";
TAB(7);"1 - Edit";TAB(7);"2 - Add";
TAB(7);"3 - Exit"
```

This will put the information on 4 separate lines because when the computer tries to perform the TAB(7) it finds that that location has been already bypassed on the present row and therefore it automatically goes to the next row. ■

MCGOVERNS XB-SKOLA, DEL 6

av Tony McGovern, Australien

VIII. KRYMPNING AV XB-PROGRAM

Vart är vi påväg i fortsättningen? Hittills har XB-skolan tittat på användardefinierade underprogram och ACCEPT AT kommandot, de två kraftfullaste egenskaperna hos TI:s Extended Basic, och även pre-scan switch-kommandot som gömmer sig i V110-manualens tilläggshäfte. Vi kommer att fortsätta med ämnen som har betydelse för användare med grundmaskinen, nämligen att krympa program för att de ska rymmas inom minnesutrymmet, och att uppnå maximal hastighet med XB. Jag kan inte se någon mening med, och har även mindre intresse för att skriva om t.ex. ljud eller sprites, eftersom dessa är bra beskrivna i manualen och varit ämnet för många böcker och artiklar. Detta betyder inte att kluriheter i användningen av dessa inte dyker upp från tid till annan.

Nog med utflykter och tillbaka till verkligheten. Låt oss nu titta på hur man ska möta budskapet 'MEMORY FULL'. Detta uppträder i varierande former även när du har expansionsminne med totalt 48 kbytes RAM. Program tycks alltid behöva mer minne än som finns tillgängligt! Jag känner en viss tvekan till att diskutera detta ämne eftersom många åtgärder som ingår i att krympa ett program annars endast kan betraktas som dålig programmeringsteknik, då de gör raderna dunkla och svåra att modifiera och vidareutveckla. Den andra stora nackdelen som måste beaktas när programmet krymps är snabbheten vid körningen. Om man programmerar för största snabbhet blir programmet vanligen längre än annars. Det enklaste exemplet som visar detta är att rulla ut en kort loop som upprepas ett bestämt antal gånger. En FOR-NEXT loop ger kompakt kod men medför nackdelen med overhead som kunde undvikas genom att skriva ut innehållet i loopen det önskade antalet gånger. Ämnet kodning för största snabbhet vid programkörningen kommer att behandlas senare i XB-skolan, medan snabbhet som offrar kompakt kod endast kommer att behandlas i förbigående. Ju mer omfångsrikt språket är desto mer möjligheter finns det att optimera koden på det ena eller andra sättet. Konsol-Basic erbjuder mycket färre sätt att göra detta än Extended Basic och är inte lika roligt.

När ska du bry dig om att försöka krympa koden? Kom ihåg att XB endast kan ladda ett program itaget med OLD eller RUN, så utöver laddningstid från kassett, eller skivutrymme, finns det inget skäl att krympa ett program som kan köras i det minsta minnesutrymme det är avsett för. De flesta användare med flexskivor har 32 kbytes expansionsminne, så detta betyder den nakna grund-datorn (konsolen). Basic-program som ska lagras i Mini Memory modul-RAM är de enda som verkligen behöver göras ännu mindre. Om du inte från början vet med dig att du kommer att få brist på minnesutrymme på grund av stora numeriska vektorer/matriser, eller behov av maximalt utrymme för att lagra strängar, var frikostig med att utförligt dokumentera ditt program med REM, använd många underprogram (CALL), använd lättförståliga variabelnamn, undvik att återanvända variabler för andra ändamål ..... och sedan får du brist på minnesutrymme.

För det första måste ett program vara tillräckligt kort för att kunna laddas. Detta beror endast på programmets längd. För det andra måste det kunna klara av pre-scan vid körningen. För att pre-scan ska kunna lyckas måste det finnas tillräckligt utrymme över för variabelpekare och tabeller för underprogram, och utrymme avsätts för numeriska variabelvärden med 8 bytes per tal. Strängvariabler tilldelas inte utrymme förrän det verkligen behövs, så det är möjligt att ett program kraschar senare på grund av brist på minnesutrymme för strängar. Den välkända hickan hos långa Basic-program inträffar när Basic letar efter och återtar strängutrymme när den har slut på nytt utrymme. XB gör samma sak, men den har mycket snabbare

'garbage collection'. Låt oss nu titta på hur man kan krympa ett program, genom att börja med saker som endast påverkar programmets längd.

Den mest närliggande åtgärden är att ta bort REM från ditt program. Jag rekommenderar att detta lämnas till senare i utvecklingsprocessen eftersom du placerat dem där som hjälp. Behåll dem åtminstone tillsvidare. Om du har följt XB-skolans tidigare råd att använda många lättförståliga underprogram så behöver du inte många REM. Av samma skäl ska du inte i detta skede förkorta underprogramnamn så att de ej känns igen. Basic som är ett interpreterande språk, där källkoden även är koden som körs, har problemet att kommentarer inte tas bort av en kompilator eller assemblerare och konkurerar om minnesutrymmet med programkörningen. Ett sätt att kringgå detta problem är att bibehålla REM i en kopia av filen efter det att utvecklingen är klar, även om detta gör koden för lång för att kunna köras. Dessa REM kan alltid tas bort senare.

Nu är det dags att titta på vad som gör ett XB-program så långt som det är. Till att börja med låt oss titta på två mycket korta program för att rensa skärmen.

```
100 CALL CLEAR
```

Innan du knappar in detta ska du rensa maskinen med NEW och prova SIZE. Knappa sedan in detta program och skriv SIZE igen. Skillnaden är längden av programmet 13928-13914 = 14. Jag kommer i huvudsak att använda SIZE för grunddatorn för enkelhets skull, men det finns några intressanta skillnader. Med expansionsminne kommer XB att lista det höga minnet (24 kbytes) och stacken var för sig, och överhuvudtaget inte bry sig om det låga minnet (8 kbytes). Extended Basic lagrar programmet och numeriska variabler i höga minnet (24 kbytes), medan stacken - 12 kbytes av VDP-minne - innehåller variabelbeskrivningar, underprogram-detaljer, PAB (diskbuffert o.dyl.) och utrymme för lagring av strängar. Allt detta måste rymmas inom 14 kbytes av VDP-RAM med XB/grund-datorn. Konsol-Basic använder överhuvudtaget inte expansionsminnet (ens när det finns anslutet). Prova nu ett andra program som nästan gör samma sak

```
100 DISPLAY ERASE ALL
```

och skriv SIZE. Nu behövs endast 9 bytes! Trots att listningen av detta program är längre, tycker datorn att det är kortare. Titta i XB-manualen sid 40 där du kan se att alla tre orden DISPLAY, ERASE, ALL är uppräknade som reserverade ord, liksom CALL men inte CLEAR. Reserverade ord behandlas annorlunda -- när du knappar in raden översätts de till en symbol med ASCII-värden över 127. 'CLEAR' behöver 7 bytes, en för symbolen för sträng, en som längdbyte, och 5 för själva strängen. Varför används symboler? Det förkortar programmets längd, och gör det även lättare för interpretatorn att känna igen dem vid programkörningen. Extended Basics uppsättning av symboler är mycket begränsad och inbyggda underprogram är det sätt på vilket XB kringgår detta.

Du är inte tvungen att enbart lita på mitt ord. Om du har ett utbyggt system så kan du skriva program som använder CALL PEEK för att utforska lagrade program, eller ännu bättre använda E/A DEBUG (assemblerat på nytt som okomprimerad objektskod så att XB CALL LOAD kan hantera det) för snabbare titt. Med konsol & XB kan du som bäst få en indirekt visshet genom att knappa in <CTRL+olika tangenter> i ett REM-kommando och sedan lista detta. Var försiktig, du kan krascha datorn på beundransvärda sätt genom detta. Någon glömde bort att upplysa datorn om att inte försöka att översätta symboler tillbaka till reserverade ord när REM listas. Har du lagt märke till att när du knappar in filbeskrivningar så kommer vissa reserverade ord som även fungerar för andra ändamål att listas med ett extra mellanslag, medan andra inte. Mini Memory EASY-BUG tillåter även att du tittar direkt i VDP-RAM eller modul-RAM för att se Basic-program i dess interna tillstånd.

I TI Basic, till skillnad mot de dialekter som sparar program som ASCII-filer, lagras alltid radnumret som ett 2 bytes heltal, och program-

längden blir lika vare sig rad #1 eller rad #10000 används. Prova olika radnummer i något av ovanstående exempel. Om du inspekterar programmet med PEEK, räkna inte med att hitta radnumret i början av programraden. Det finns i en fristående tabell nedanför programmet, och var 4:e byte har radnummer följt av adressen till själva programraden. Radnummertabellen sorteras i ordning, medan nya eller editerade rader alltid läggs till den lägre adressidan av programblocket. Själva programraderna föregås av en längdbyte och avslutas med en noll-byte (>00). Jag kommer inte att gå in på detta här men du kan använda denna allmänna information för att tolka de olika tidsfördröjningarna när du editerar eller matar in en ny rad.

Av detta kan du se att det finns en overhead på 6 byte tillhörande varje nytt radnummer. Knappa nu in ovanstående programrader som rad #100 och rad #200 och skriv SIZE. Kombinera dem sedan till en enstaka rad

```
100 CALL CLEAR :: DISPLAY ERASE ALL
```

och prova SIZE igen. Du sparar 5 bytes. Det reserverade ordet "::" har kostat 1 byte, medan 6 bytes har sparats genom att ha en rad mindre. Om du nu krymper ett 500 raders konsol-Basic-program till 200 XB-rader med flera kommandon per rad så har du vunnit 1500 bytes. Naturligtvis kan du inte göra detta med varje rad eftersom radnummer, förutom att vara markeringar för radeditorn, anropas med GOTO och GOSUB, så du får vanligen ett fåtal korta rader som du inte kan krympa. FOR -NEXT slingor fungerar mycket bra inom eller mellan rader med flera kommandon. Användningen av pre-scan switch-kommandot kostar på eftersom du måste ha !@+ och SUBEND på olika rader i slutet av varje berört underprogram. Trots detta är det vanligen värt att göra det även om ett långt program får flera hundra bytes uppbundna av pre-scan switchning. Vid problem på slutet kan du alltid ta bort pre-scan switchningen med början från det kortaste underprogrammet.

Hur stor plats upptar en variabel? Tag en enkel numerisk variabel. Det åtgår 8 bytes för radix-100 flyttalsformat som används av både konsol-Basic och XB för alla tal (de gör t.o.m. 1+1 med 14 signifikanta siffror varje gång - ytterligare ett skäl till att de är långsamma). För det andra måste interpretatorn kunna hitta var detta värde är lagrat så det behövs 2 bytes för pekare till värdet, och ytterligare 2 för att peka på namnet tillhörande detta värde. Dessutom måste den lagra typen av variabel, om den är numerisk eller sträng, enkel eller vektor/matris, skapad med DEF eller normal. Även ett Basic-språk som tillåter långa variabelnamn kan behöva en längd-post, även om det inte är helt tvingande. Allt som allt behövs det som minimum 14 bytes av overhead för varje enkel numerisk variabel.

Som jag har påpekat i andra sammanhang i denna artikelserie, har TI på sitt hemligt självförgörande sätt, aldrig uttryckligen avslöjat detaljerna. TI Basic är ganska säkert mycket entydig när det gäller detta från fall till fall, eftersom varje konsol kan fås att arbeta med olika E/A- eller Minimem-Basic tilläggsfunktioner som NUMREF. I motsats till detta består varje XB-modul av sin egen uppsättning av tilläggsfunktioner, och behöver endast vara i överensstämmelse med sig självt (XB är äldre än konsol-Basic, 1980 resp 1981 ö.a.). Det finns information i TI:s publicerade data (XB, E/A & tekniska manualer), som ger detaljer om den VDP-stack som byggs upp av E/A CALL LINK med några antydningar hur det ska ändras för XB-versionen. Så för att använda XB CALL LINK på denna detaljeringsnivå måste du arbete med antydningar. Detta görs nu från tid till annan, men TI tycks inte uttryckligen ha garanterat att sådana åtgärder skall fungera med alla XB-moduler, eller att LINK-stacken liknar interna tabeller. Det är mycket sannolikt att så är fallet. Endast TI vet med säkerhet. Sedan gjorde TI stora förluster medan Apple och IBM tjänade mycket pengar på att vara mer öppna om sina maskiner, även om Apple tycks gå mot mer hemliga vägar vartefter det blir äldre och mer oförstående.

Dax igen för några små program-försök. Knappa in detta lilla program

```
100 A=0
```

Innan du gör något mer fundera ut hur många bytes detta använder. Svaret är 11. Vid inmatningen har radeditorn redan förstått att 'A' är en variabel (eftersom den börjar med ett godkänt tecken) och inte ett reserverat ord och det representeras exakt som det uppträder utan symbol ("token"). Å andra sidan bryr det sig ännu inte om att '0' är avsett att bli ett tal och behandlar det som en sträng. Om det inte är ett äkta tal, som t.ex. 2N, upptäcks detta först senare när det körs med RUN och försöker att omvandla det till ett flyttal.

Prova SIZE av programmet, kör det med RUN och prova SIZE igen. XB återställer inte allt förrän du editerar, som du känner till från avlusning efter BREAK (FCTN-4) av programkörning. I detta skede får du mer information från ett utbyggt system, som kommer att visa att 8 bytes minne har använts och 9 bytes av stackutrymmet. Upprepa sedan detta med ett längre variabelnamn. Längden återspeglas både i det ursprungliga programmets längd och i det använda stackutrymmet. Stackutrymmet använder 2 bytes plus variabelns namnlängd utöver det minimum vi tidigare sett. Dessa 2 bytes används troligen för en länkad liststruktur för att underlätta sökning i tabellen, och det finns en symbollista av variabelnamn. Stäng nu av expansionssystemet och var som alla andra med enbart grund-datorn, och upprepa ovanstående. Du kommer nu att finna att ökningen utöver programlängden alltid blir 14 bytes som vi visat tidigare oberoende av längden av variabelnamnet. Prova nu

```
100 A23456789012345,
    A23456789012345=0
```

Fortfarande 14 bytes extra overhead efter RUN! Ändra det andra A till ett B för att göra ett nytt variabelnamn, 15 bytes långt. Endast ytterligare 14 bytes overhead! Hur hänger det ihop? Kanske klarar den sig utan den fullständiga länkade ordlistan och krymper det hela något, men hur är det med symboltabellen? Den enda riktiga slutsatsen är att det inte finns någon sådan, utan pekar på det första ställe i programmet där variabelnamnet upptäcks av pre-scan. Läs igenom XB-skolan om pre-scan ytterligare en gång. XB söker alltid i VDP-RAM efter variabelnamn även om själva programmet och utpekade numeriska värden lagras i expansionsminnet.

Om du vill göra en snabbare interpreterad Basic bör du, genom pre-scan, byta ut alla variabelnamn mot någon symbol tillsammans med pekare till lagringen för att eliminera sökning i tabell. Det är just detta som TI hävdat att man gjort i sin Software Development Handbook med Basic för sina 990 minidatorer. Olyckligtvis misslyckades de med att göra en bra maskin av 99/4.

Nästa XB-skola kommer att fortsätta med principerna för krympning av program, och mera komma in på programskrivning. ■

PARSEC

by Tony McGovern, Australia

A few weeks ago an obscure IEEE journal arrived. I never did order it and I think it just comes as a consequence of some others I do subscribe to. There on the front page contents was PARSEC - Process Analysis etc etc to complete the acronym. I haven't seen that up and running for several years now, but the module was a major point in the history of the TI-99/4a. Now a name like that leaps to the eye so I had a look inside. Sure enough the article was written by a bunch of engineers at the TI wafer fab facility in Lubbock. Must be some tribal memories there though none of the biographies admitted to have ever worked on the Home Computer. ■

FUNNELWEB LOAD&RUN

Nya versioner kräver att du suddar inmatningsfältet efter sista filnamn annars blir det ERROR pga pathname.■

ORDBEHANDLING MED 99/4A

av Jan Alexandersson

Du använder en ordbehandlare för att skriva in texter i datorn. Du kan sedan enkelt lägga till och ändra i texten. Resultatet kan lagras på flexskiva (i vissa fall även kassett) och skrivas ut på en skrivare.

ORDBEHANDLARE FÖR XB

Det finns många olika ordbehandlingsprogram till TI-99/4A. Sådana program skrivna i Extended Basic har publicerats i olika utländska tidningar men även i Programbiten. Lennart Thelander har tidigare beskrivit sin LT-Writer för Extended Basic med 32 kbytes expansionsminne (fungerar med kassett eller flexskiva) och LT-MiniWriter för enbart XB. L-E Fjellstedt har beskrivit sin TEX som är en förbättring av ett program från 99'er magazine dec 1982.

TI-WRITER

Det mest använda ordbehandlingsprogrammet är TI-Writer som består av en modul, en flexskiva och en manual på 176 sidor i en stor ringpärm. Enbart manualen är värd det pris du idag kan köpa den för i USA. TI har släppt en uppdaterad flexskiva (version 2) som finns i programbanken. TI-Writer har senare vidareutvecklats av Tony McGovern till Funnelweb 4.30 och R.A.Green till RAG-Writer 4.5.

Alla dessa TIWR-liknande ordbehandlare lagrar texten i DIS/VAR 80-filer på flexskiva. Varje rad lagras som en post. Den sista posten består av TAB-inställningar för ordbehandlaren. Denna TAB-rad syns inte med TI-Writer men du kan se den om du tittar på filen med DM 1000 och T(ype). TI-Writer version 2 på uppdateringsskivan skapar filer som inte kan laddas med version 1, Funnelweb eller RAG-Writer. En sådan TAB-rad kan tas bort med ett enkelt XB-program:

```
100 REM REMOVER XB
110 INPUT "FIL ":F$
120 OPEN #1:"DSK1."&F$,INPUT
130 OPEN #2:"DSK2."&F$,OUTPUT
140 LINPUT #1:A$
150 IF LEN(A$)=0 THEN 170
160 IF ASC(A$)>127 THEN 180
170 PRINT #2:A$
180 IF EOF(1)THEN 200
190 GOTO 140
200 CLOSE #1 :: CLOSE #2
210 GOTO 110
```

Du kan även ta bort TAB-raden genom att välja PF (Print File) och skriva DSK1.FILNAMN istället för PIO. Det finns en bug i DSRLNK för TI-Writer version 2 som gör att den inte fungerar med två samtidiga diskkontrollkort. Jag har Myarc HFDC på CRU >1000 och TI diskkontrollkort på CRU >1100 så jag har ingen möjlighet att använda version 2. Ovanstående XB-program ordnar så att jag ändå kan ladda in filen i Funnelweb.

SVENSKA BOKSTÄVER

TI-Writer version 2 har en TXTE-fil som som har svenska kommandon och en CHARA-fil som har svenska bokstäver i texten. Jag tycker att svenska kommandon är meningslösa eftersom de engelska LF (Load File) och SF (Save File) duger mycket bra. En speciell svensk CHARA-fil är dock nödvändig. Uppdateringsskivan innehåller CHARA-filer för många olika språk inkl. svenska. Den filen kan alltid användas men den måste döpas om till C1 för Funnelweb och CHARA1 för RAG-Writer.

Jag har själv gjort en förenklad svensk version som enbart har följande svenska tecken: 91 Ä, 92 Ö, 93 Å, 123 ä, 124 ö och 125 å (även 96 é borde kanske tas med). Det blir då något enklare att se Basic-program i ordbehandlaren. När du skriver in svensk text så använder du den tangent som motsvarar respektive ASCII-kod (se ditt Reference Card för Basic). Du använder hakparenteser m.m. som du ganska snart lär dig att hitta som svenska tecken. Du ser ju

alltid på skärmen om du har tryckt på rätt tangent.

Denna sorts CHARA-fil är enbart ett hjälpmedel när du skriver in texten. Vid utskrift på din skrivare är den till föga hjälp. Du måste då knappa in kontrollkoder med CTRL-U som ställer om skrivaren till svenska tecken. Jag har en Epson-kompatibel NEC P6 skrivare som behöver ESC R (5). Jämför med din egen printer-manual och korrigera för eventuella skillnader. Du knappar in det så här:

CTRL-U FCTN-R CTRL-U R CTRL-U SHIFT-E CTRL-U

Detta motsvarar i Basic:
CHR$(27);"R";CHR$(5)

På min skrivare finns 12 olika språk där n i ESC R (n) kan varieras beroende på språk:

n	CTRL-U	Språk
0	SHIFT-@	USA
1	SHIFT-A	Frankrike
2	SHIFT-B	Tyskland
3	SHIFT-C	England
4	SHIFT-D	Danmark I
5	SHIFT-E	Sverige
6	SHIFT-F	Italien
7	SHIFT-G	Spanien
8	SHIFT-H	Japan
9	SHIFT-I	Norge
10	SHIFT-J	Danmark II
11	SHIFT-K	Nederländerna

När du skriver ut engelsk text bör du normalt välja USA som motsvarar CHR$(27);"R";CHR$(0) i Basic.

VÄNSTER- OCH HÖGERMARGINAL

Du vill nästan alltid ha en marginal för hålslagning av pappret. Det finns tre olika sätt att ordna detta vid utskriften:

- kontrollkod som för Epson blir CHR$(27);"l";"CHR$(n) där du väljer avståndet från vänsterkanten. n=10 kan ofta vara lagom. Du knappar in detta med CTRL-U FCTN-R CTRL-U l CTRL-U SHIFT-J CTRL-U

- TAB inställningar som väljs med BACK (FCTN-9) och sedan T. Du kan markera L (Left margin) för vänster-marginal på TAB-linjen och även R (Right margin) för högermarginal.

- Formateringskommandon .LM (Left Margin) och .RM (Right Margin) som alltid måste föregås av .FI (Fill). För dig som har en vanlig 99/4A med 40-kolumnsskärm är detta ofta den bästa lösningen. Skriv in texten med vänstermarginal 0 och högermarginal 39. Formateraren kan sedan skriva ut detta med önskad bredd, t.ex. 55, 65, 75 eller 80 kolumner. Om du vill ha mer än ett mellanslag måste detta markeras med hårda mellanslag ^^^^^^ eftersom .FI annars packar texten så mycket som möjligt med endast ett mellanslag.

ÖVRE OCH UNDRE MARGINAL

Du kan alltid använda formateraren för utskrift även om du inte har lagt in några speciella forma-teringskommandon. Vid utskrift får du då automatiskt en marginal upp-till och nertill. Utskriften börjar på ny sida när den första är full. TI-Writer har som default-värde 66 raders sida enligt amerikansk standard. När 60 rader har skrivits ut ger programmet Form Feed (ASCII 12) så att utskriften börjar på ny sida. I Sverige används normalt 12" papper med 72 rader (förlängd A4). Du kan informera formateraren om detta genom .PL 72 (Page Length). Detta ger förutom marginalerna 66 rader text.

SPECIALTECKEN

Följande tecken kan inte användas fritt tillsammans med formateraren:

- & tecken 38: understrykning
- @ tecken 64: fet stil
- * tecken 42: adresslista
- ^ tecken 94: blanktecken
- . (punkt) i början av en rad

& kommer att ge understrykning av efterföljande ord. Om du verkligen vill skriva ut ett enstaka & så måste du skriva &&. @ ger på motsvarande sätt fet stil genom att texten skrivs fem gånger. Du kan tvinga fram ett @ med @@.

* kan inte användas tillsammans med siffror och något botemedel mot detta finns inte.

^ tvingar fram blanktecken även om många anges. Man kan kringgå detta med .TL 94:94 som släpper ut taktecknet (eller tyskt Ü) som ett vanligt tecken.

. (punkt) i början av en rad uppfattar formateraren som ett formateringskommando oberoende av vad som står efter punkten så efterföljande rad skrivs inte ut. Du kan kringgå detta genom att skriva en meningslös kontrollkod före punkten. Jag brukar använda FCTN-U SHIFT-T FCTN-U som återställer till normal teckenbredd. Sådana kontrollkoder räknas inte som bokstäver av printern vid utskriften så punkten kommer längst till vänster. Själva TI-Writer räknar den dock som ett tecken vilket kan krångla till efterföljande tabeller eller spalter.

ATT SKRIVA TEXT

Om du ska skriva en enklare text som t.ex. en artikel i Programbiten gör du så här. Börja med att ställa in TAB-raden med högermarginal (R) som skall vara 36 i tidningen. I annat fall väljer du 39 med 99/4A eller 65 med 9938. Använd inga formateringskommandon och skriv texten så att den kan skrivas ut direkt från editorn med PF (Print File). Använd inga dubbeltecknade && eller @@. Använd inte ^^^^^^ för mellanslag.

Skriv texten löpande utan att trycka på ENTER i slutet av raden. Fortsätt att skriva tills stycket är slut och avsluta då med ENTER och ytterligare ett ENTER för att få en tom rad före nästa stycke. För att underlätta för REFORMAT (CTRL-2) bör du göra ett mellanslag före radmatningen (CR). Samla ihop ett antal stycken till ett kapitel som du skriver en rubrik före.

TANGENTBORDSMALL

Du måste även ha en tangentbordsmall för TI-Writer som finns i PB 88-4. Den undre raden används med FCTN + siffra och den övre raden anger CTRL + siffra. Jag har kvar några mallar från SouthWest Ninety-Niners som du kan få för 22 kr. Kontakta mig i förväg eftersom jag inte kommer att skaffa nya när de tar slut. Du får samtidigt en 8 sidors kopia på deras användartidning i Tucson, Arizona. Mallen består av fem hopsatta mallar av plastöverdraget styvt papper för följande moduler: konsolen, TI-Writer, Multiplan, Editor/Assembler, Terminal Emulator II. Du kan klippa isär mallen i fem delar men jag rekommenderar att du behåller den odelad.

MANUAL

För att kunna använda TI-Writer, Funnelweb eller RAG-Writer på bästa sätt så behöver du en ordentlig manual. Det bästa är att skaffa TI:s originalmanual som följer med TI-Writer även om du enbart tänker använda Funnelweb. Det finns även skivor i Programbanken med förenklade manualer där det viktigaste finns beskrivet.

REFERENSER

PB 85-1 TI-Writer/Multiplan upgrade
PB 85-1 LT-Writer
PB 85-1 LT-MiniWriter
PB 85-4 LT-Writer, modifierad
PB 86-3 Korrekturläsning Spell Check
PB 87-2 Fel i TI-Writer version 2
PB 87-2 Texteditor i c99
PB 88-4 TEX ordbehandlare
PB 88-4 Tangentbordsmall EA & TIWR
PB 89-1 Funnelweb
PB 90-1 TI-Writer once again
PB 90-3 RAG-Writer 4.5 Formateraren

Micropendium
maj 84: Intelpro Companion
sep 84: Dragonslayer Spell Check
okt 85: Navarone Console Writer
feb 86: Paolo Bagnaresi BA-Writer
jul 86: McGovern Funlwriter 3.3
jun 87: Corcomp Writerease
maj 88: Jack Sughrue PLUS!
okt 88: Corcomp Writerease Update
apr 89: R.A.Green RAG-Writer
feb 90: Chicago TI-Writer Supplement■

ASGARD ROCK RUNNER

Ett mycket bra spel kan köpas från Asgard Software. Pris USD 13 + flygporto USD 3 (MicroP feb och maj 90).■

ASSEMBLER TUTORIAL - 1

by Tony McGovern, Australia

HERE, THERE or ANYWHERE

The simplest possible form of assembly program code to think about is code that works at a fixed address for each instruction and data word. Every part of the program knows exactly where every other part is, at absolutely fixed addresses. This is known as "absolute" code and is all that could be written on many microcomputers more primitive than the TI-99/4a, or with TI's cassette based Line-by-Line assembler for the Minimem module. Now by the time an E/A program file is prepared it ends up in this form, and usually must be loaded and executed at just the address specified in the 3 word file header.

What I intend to do in this article, or short series depending on how the length goes, is to look at how the TI-99 system allows the programmer to write a single piece of code that may be executed at various addresses. Assemblers for some newer micros such as the Motorola 68000 series in fact make it very difficult to write purely absolute code (though the ingenuity of programmers can never be underestimated).

So let's look at the possibilities in increasing order of complexity. In TI-99/4a systems, code is normally produced by the Assembler in the form of Tagged Object code. Though the 9900 may now itself be old history, the TI tagged object format lives on, not being restricted to 9900 code but used to express for example TMS320 series digital signal processor (DSP) code. Consult your E/A manual for full details as I don't really want to go into all that here. In fact the TI-99 assembler is a direct carry-over of a mini-computer assembler, and supports more facilities than the E/A loaders will handle.

Suffice it to say that in tagged object code, each byte or word (compressed format) of code, data, or address information is preceded by a tag character which the loader reads as an instruction on what to do with the word. The tag characters are nothing more than 1..9, A..., and it is the job of the loader to keep track of what is a fixed data word or instruction opcode (they are the same thing to a loader) and what is an address reference that needs to be adjusted at load time. In ordinary or uncompressed object code the words are written out in hexadecimal and the Dis/Fix 80 files can be read and edited with a text editor. Compressed object code keeps the tags as alphabetic characters but expresses the word after each tag as 2 machine binary bytes. The only practical way to read or edit this is to use a sector editor on disk in it's hexadecimal display mode. Tags "9" and "B" indicate absolute addresses and data or instructions, and "A" and "C" indicate relocatable.

Absolute code is produced if an AORG directive is used in the assembly source code. You are instructing the loader to place the code exactly as is, at the definite address specified. The TI system however is biased towards the use of relocatable code especially in conjunction with Basic or XBasic. If no specific directive is given the assembler produces tagged object code that the linking loader recognizes as relocatable. The loader then decides where it can put the code, and adjusts the relocatable addresses and data accordingly as it puts them in place. Once it is loaded you cannot tell the difference between this code and data in memory and the code and data in memory if it had been placed there by a absolute load which happened to be to the same address. Only the loader knows and it keeps track only of relocatable programs so it can figure where to put the next relocatable block.

All of the above is familiar stuff for any assembly programmer. What we really want to look at is how to get code to execute correctly at

addresses other than those at which it was initially loaded. Why on earth would you want to do that ? One reason is that you may wish to have code which is placed on one or repeated occasions to execute elsewhere, commonly in the PAD for speed or to avoid memory bank-switching traumas as with Myarc RAM-Disks.

There are two ways to generate such code. One is to trick the assembler into generating code that runs correctly at the address at which it is ultimately placed, rather than at the address at which it is initially loaded. The assembler itself provides the way with the DORG (for Dummy ORiGin) directive. The more thorough-going way is to write code which genuinely does not care which address it is at, known as position independent code. This often turns out to be very easy for short pieces of code and rather more tricky for typical longer program segments.

First we will look at the DORG directive. When the assembler encounters this it quits producing any code but still keeps track of addresses following the DORG statement. When it next encounters a AORG or RORG it goes back to producing code. The usual examples you will find in TI source code are to do with setting up data fields or workspace register names in areas of memory not included in the bounds of the program. Simple cases like these can also be handled as EQUates.

```
*  Address generated by DORG

       DORG >8328
BUFMWS DATA 0          Wsp & R0
TBLND  DATA 0          R1
EDBFND DATA EDITBF     R2
       DATA SQSBUF     R3
LINEN  DATA 0          R4
       RORG


*  Addresses by EQUates

BUFMWS EQU >8328     Wsp & R0
TBLND  EQU >832A     R1
EDBFND EQU >832C     R2
LINEN  EQU >8330     R4
```

The two segments of assembly source above both do the same job of defining addresses for a workspace to be set up in PAD RAM. As far as the rest of the program is concerned both pieces of code have exactly the same effect in setting up addresses for part of a workspace for use by the program in PAD. At that level of complexity there is not probably much to choose between them, but the first one is better programming practice - for the programmer - the assembler doesn't really care. The second code causes 4 entries in the symbol table. So does the first code, and the assembler has to do more work to get that done. Why then is it better ? The reason is that it shows more clearly what is intended for this workspace, and very succintly and clearly reminds the programmer of initialization code that will need to be executed. Remember that though there are data statements in the DORGed code, no object code is generated that would place data at those addresses. That code has to be written separately in either case as part of the program.

If the DORGed code were rather more complex it would become a real pain to to keep track of where the EQUates should go. It is much easier just to make a copy of the code using the editor. Let us look how to generate a block of code for the DORG directive from the actual code which has to be carried in at one address and later moved and executed at the address set up by the DORG directive. An example of that is in the Funnelweb Assembler, where the filename entry code has to be brought in latched onto the end of the program file AT (to save slowing the loading and to save wear and tear on disk drives if another program file were generated), but for various reasons needs to execute elsewhere in memory.

This is done by having one copy of the code DORGed at it's actual execution address to establish the labels, and a second modified copy for which the assembler generates code in the main block. One way or another the program shifts the code as a block to it's execution address. Some care however has to

be taken with labels. Addressing of a memory location is done in three ways in the 9900 processor. The shortest is register addressing, R0 through R15, which at 4 bits is short enough to have two of them in one instruction. The processor goes to memory at a full word address obtained by adding this offset to the Workspace Pointer. This does not really care about where the code is executing from as long as there is no conflict. The 9900 family is different from most other micros in this regard, though the approach is re-appearing in various forms in some RISC processors. The next shortest is Program Counter (PC) relative addressing as found in jump instructions or for data words in Immediate instructions such as LI R0,>100. In the JMP type of instruction 8 bits are used as a word offset counter from the word following the JMP or other such instruction. This one is inherently dependent on the current PC value, and if the JMP is to an assembly label, this must be in the right place relative to the actual execution address. The last of course is just the normal full 16-bit address which takes up a whole word.

So we have a DORGed copy of the code which has the address labels (well, almost all of them) but generates no actual code, and the copy which generates the code but needs only local PC relative target labels for Jump instructions. It is a good idea to use dummy labels for other addresses in the code for clarity. The code example is for illustration only.

```
*  Copy for address generation
*
       DORG DADDR
ENTCOD EQU  $
       MOV  R11,R10
       BL   @SUBR1
       LI   R0,>20
ENT10  MOV  *R3+,@VDPWD
       DEC  R0
       JGT  ENT10

       B    *R10

SUBR1  EQU  $
       ..
       ..
       RT

       RORG ( or AORG )

*  Program continues


*  Copy for code generation
*
CODSEG EQU  $
       MOV  R11,R10
       BL   @SUBR1
       LI   R0,>20
ENT10X MOV  *R3+,@VDPWD
       DEC  R0
       JGT  ENT10X

       B    *R10

SUBR1X EQU  $
       ..
       ..
       RT

CODLEN EQU  $-CODSEG
```

The example may be short and artificial, but it does show all the essential features. The BL instruction uses a normal full word address as DORG generates the address for the SUBR1 label at it's correct ultimate location in the DORG block. In the regular segment the BL must also be to SUBR1 and the SUBR1X label is for information only and in fact may be omitted entirely, at the cost of clarity in the code (while you could BL to anywhere in the DORGed BL as long as the instruction length remains the same). The next label in the DORGed code is a PC relative jump target. For clarity we put it in the right place, but it doesn't really matter (and even a JGT $ could be used). Over in the actual code the ENT10X label is the one that really sets up the correct jump on JGT. If you forgot to change the name of the label in the JGT instruction, the assembler will generate an Out of Range error. It is better to use a new label for the local Jxx rather than to count out the offset and use JGT $-6 as the presence of the label makes it easier to compare the DORGed code with the RORG or AORGed version. If you have an 80 column system (AVPC or Mechatronics) then the split-screen dual file View function of

DiskReview is very handy in tracking down possible problems.

If you had gotten carelessly over-enthusiastic and used SUBR1X as the label for the BL in the generated code, then the assembler would not flag an error. This can be an insidious problem in code testing, since the original copy of the code might still be in place early in program operation and only corrupted later by data or whatever. Finally VDPWD is a full word address independent of the DORGed block. As a last reminder if the code is changed, matching changes must be made to both copies.

Next installment will be about writing strictly position independent code for the 99/4a and on some precautions for writing subroutines so that they may be called from any workspace.

Funnelweb Farm
May 30th / 1990 ■

OTHELLO – SPEL FÖR XB

av Per Virving

```
1 CALL CHAR(91,"00280038447C
444440028007C4444447C00382838
447C4444")
2 CALL INIT :: CALL LOAD(-31
806,16)
10 CALL CHAR(128,"187EFFFF7E
18",140,"187EFFFF7E18")
20 CALL COLOR(13,2,1,14,16,1
)
100 FOR KILO=0 TO 12 :: CALL
 COLOR(KILO,16,1):: NEXT KIL
O
110 CALL SCREEN(13)
120 DEF CTR$(X$)=SEG$("
         ",1,14-LEN(X$)/2)&X
$
130 DEF LN$(X)=RPT$(CHR$(143
),X)
140 CALL CHAR(143,"0000FF")
150 REM
160 DIM A(9,9),I4(8),J4(8),C
$(8),D$(2)
170 CALL CLEAR :: DISPLAY AT
(3,1):CTR$(LN$(24)): : :CTR$
("- OTHELLO -"): : :CTR$(LN$
(20))
175 DISPLAY AT(12,1):CTR$("a
 program by"):CTR$("p. virvi
ng"):CTR$("1987"): :CTR$("co
pyright s.c.s"): :CTR$("ver.
 1.0")
180 DISPLAY AT(24,1):"   ins
tructions(y/n): Y"
190 ACCEPT AT(24,23)VALIDATE
("YNyn")SIZE(-1):X$
200 IF (X$="N")+(X$="n")THEN
 360
210 CALL CLEAR
220 PRINT "OTHELLO is played
 in a fieldof 8*8 squares, r
ow 1-8 and columm A-H"
230 PRINT "You always start
with this  arrangement:":"
 ";CHR$(140);CHR$(128):"   "
;CHR$(128);CHR$(140)
240 PRINT "When you've put a
 piece, youget all your riva
ls pieces  which is between
your piecesand the piece you
 put"
250 PRINT "you always have t
o take one of your rivals pi
eces, if   you can't do that
,it's the  rivals turn"
260 PRINT "(if you can't put
 any piece,you answer ""A0""
)"
360 F2=0
370 RESTORE :: PRINT :"will
I play the             best
i can(Y/N): Y"
380 S2=0
390 ACCEPT AT(23,18)VALIDATE
("YNyn")SIZE(-1):X$
400 IF (X$="N")+(X$="n")THEN
 420
410 S2=2
420 B=-1
430 W=1
440 D$(B+1)=CHR$(128)
450 D$(0+1)=" "
460 D$(W+1)=CHR$(140)
470 FOR K=1 TO 8
480 READ I4(K)
490 NEXT K
500 DATA 0,-1,-1,-1,0,1,1,1
510 FOR K=1 TO 8
520 READ J4(K)
530 NEXT K
540 DATA 1,1,0,-1,-1,-1,0,1
```

```
550 FOR K=1 TO 8
560 READ C$(K)
570 NEXT K
580 DATA "A","B","C","D","E"
,"F","G","H"
590 FOR I=0 TO 9
600 FOR J=0 TO 9
610 A(I,J)=0
620 NEXT J
630 NEXT I
640 A(4,4)=W
650 A(5,5)=W
660 A(4,5)=B
670 A(5,4)=B
680 C1=2
690 H1=2
700 N1=4
710 Z=0
720 PRINT "Do you want Black
 or white?"
730 C=W
740 H=B
750 DISPLAY AT(24,1):"W" ::
ACCEPT AT(24,1)VALIDATE("BWb
w")SIZE(-1):X$
760 IF X$="B" OR X$="b" THEN
 790
770 C=B
780 H=W
790 PRINT "Do you want to st
art? Y"
800 ACCEPT AT(23,23)VALIDATE
("YNyn")SIZE(-1):X$
810 IF (X$="N")+(X$="n")THEN
 860
820 CALL CLEAR
830 GOSUB 2670
840 GOSUB 2610
850 GOTO 1320
860 CALL CLEAR :: GOSUB 2670
 :: GOTO 890
870 IF F2=0 THEN 890
880 INPUT X$
890 B1=-1
900 I3=0
910 J3=0 :: T1=C
920 T2=H
930 FOR I=1 TO 8
940 FOR J=1 TO 8
950 IF A(I,J)<>0 THEN 1120
960 GOSUB 2270
970 IF F1=0 THEN 1120
980 U=-1
990 GOSUB 2360
1000 IF S1=0 THEN 1120
1010 IF (I-1)*(I-8)<>0 THEN
1030
1020 S1=S1+S2
1030 IF (J-1)*(J-8)<>0 THEN
1050
1040 S1=S1+S2
1050 IF S1<B1 THEN 1120
1060 IF S1>B1 THEN 1090
1070 R=RND
1080 IF R>.5 THEN 1120
1090 B1=S1
1100 I3=I
1110 J3=J
1120 NEXT J
1130 NEXT I
1140 IF B1>0 THEN 1190
1150 DISPLAY AT(22,1):"JAG M
]STE AVST] MITT DRAG"
1160 IF Z=1 THEN 1880
1170 Z=1
1180 GOTO 1320
1190 Z=0
1200 DISPLAY AT(1,1):"My tur
n:  ";C$(J3);CHR$(I3+48)
1210 I=I3
1220 J=J3
1230 U=1
1240 GOSUB 2360
1250 C1=C1+S1+1
1260 H1=H1-S1
1270 N1=N1+1
1280 DISPLAY AT(23,1):"I too
k";S1;"of your pieces"
1290 GOSUB 2610
1300 IF H1=0 THEN 1880
1310 IF N1=64 THEN 1880
1320 T1=H
1330 T2=C
1340 DISPLAY AT(24,1):: DISP
LAY AT(2,1):"Your turn:";"
                "
1350 ACCEPT AT(2,13)VALIDATE
(DIGIT,"ABCDEFGH")SIZE(2):Z$
 :: DISPLAY AT(1,11):"
  " :: DISPLAY AT(22,1):"
                       "
1355 DISPLAY AT(24,1):CTR$("
working,please wait")
1360 IF LEN(Z$)=2 THEN 1370
ELSE 1390
1370 Z1=ASC(SEG$(Z$,1,1))::
Z2=ASC(SEG$(Z$,2,1))
1380 IF Z1>47 AND Z1<57 AND
Z2>64 AND Z2<73 OR Z1>64 AND
 Z1<73 AND Z2>47 AND Z2<57 T
HEN 1400
1390 CALL SOUND(100,1000,0):
: DISPLAY AT(3,1):"- Try aga
in" :: GOTO 1340
1400 IF Z1>57 THEN 1420
1410 I=Z1-48 :: X$=CHR$(Z2):
: GOTO 1430
1420 I=Z2-48 :: X$=CHR$(Z1)
1430 IF I<0 THEN 1350
1440 IF I>8 THEN 1350
1450 IF I<>0 THEN 1540
1460 DISPLAY AT(3,1):"Do you
 give up this move? Y"
1470 ACCEPT AT(3,27)VALIDATE
```

```
("YNyn")SIZE(-1):X$
1480 DISPLAY AT(3,1)
1490 IF LEN(X$)=0 THEN 1460
1500 IF X$="N" OR X$="n" THE
N 1340
1510 IF Z=1 THEN 1880
1520 Z=1
1530 GOTO 870
1540 DISPLAY AT(3,1)
1550 FOR J=1 TO 8
1560 IF C$(J)=SEG$(X$,1,1)TH
EN 1590
1570 NEXT J
1580 GOTO 1350
1590 IF A(I,J)=0 THEN 1630
1600 DISPLAY AT(22,1):"Occup
ied square"
1610 DISPLAY AT(3,1):"-Try a
gain."
1620 GOTO 1340
1630 GOSUB 2270
1640 IF F1=1 THEN 1680
1650 DISPLAY AT(23,1):"Not b
eside my pieces"
1660 DISPLAY AT(3,1):"-Try a
gain."
1670 GOTO 1340
1680 U=-1
1690 GOSUB 2360
1700 IF S1>0 THEN 1750
1710 DISPLAY AT(23,1):"You c
an't turn anyone"
1720 DISPLAY AT(3,1):"-Try A
gain."
1730 GOTO 1340
1740 !His move OK
1750 Z=0
1760 DISPLAY AT(2,1)
1770 DISPLAY AT(3,1)
1780 DISPLAY AT(23,1):"You t
ook";S1;"of my pieces"
1790 U=1
1800 GOSUB 2360
1810 H1=H1+S1+1
1820 C1=C1-S1
1830 N1=N1+1
1840 GOSUB 2610
1850 IF C1=0 THEN 1880
1860 IF N1=64 THEN 1880
1870 GOTO 870
1880 CALL SOUND(500,1000,0)
1890 DISPLAY AT(4,1)SIZE(11)
:"You have";H1 :: DISPLAY AT
(5,1)SIZE(6):"pieces" :: DIS
PLAY AT(6,1)SIZE(9):"I have"
;C1 :: DISPLAY AT(7,1)SIZE(6
):"pieces"
1900 IF H1=C1 THEN 1940
1910 IF H1>C1 THEN 1960
1920 DISPLAY AT(23,1):CTR$("
I  W O N ! !")
1930 GOTO 1970
1940 DISPLAY AT(23,1):CTR$("
D E A D  P L A Y ! !")
1950 GOTO 2140
1960 DISPLAY AT(23,1):CTR$("
Y O U  W O N !!")
1970 C1=C1-H1
1980 IF C1>0 THEN 2000
1990 C1=-C1
2000 C1=(64*C1)/N1
2010 IF C1<11 THEN 2130
2020 IF C1<25 THEN 2110
2030 IF C1<39 THEN 2090
2040 IF C1<53 THEN 2070
2050 DISPLAY AT(24,1):CTR$("
Perfect play!")
2060 GOTO 2140
2070 DISPLAY AT(24,1):CTR$("
tolerably well")
2080 GOTO 2140
2090 DISPLAY AT(24,1):CTR$("
a good fight")
2100 GOTO 2140
2110 DISPLAY AT(24,1):CTR$("
skilful!!")
2120 GOTO 2140
2130 DISPLAY AT(24,1):CTR$("
you cheated!!")
2140 DISPLAY AT(22,1):"one m
ore play? Y"
2150 ACCEPT AT(22,16)VALIDAT
E("YNyn")SIZE(-1):X$
2160 IF X$<>"Y" AND X$<>"y"
THEN 2230
2170 DISPLAY AT(1,1)
2180 DISPLAY AT(2,1)
2190 DISPLAY AT(3,1)
2200 DISPLAY AT(22,1)
2210 DISPLAY AT(23,1):: CALL
 CLEAR
2220 GOTO 370
2230 DISPLAY AT(22,1)
2240 DISPLAY AT(23,1)
2250 !
2260 GOTO 2760
2270 FOR I1=-1 TO 1
2280 FOR J1=-1 TO 1
2290 IF A(I+I1,J+J1)=T2 THEN
 2340
2300 NEXT J1
2310 NEXT I1
2320 F1=0
2330 RETURN
2340 F1=1
2350 RETURN
2360 S1=0
2370 FOR K=1 TO 8
2380 I5=I4(K)
2390 J5=J4(K)
2400 I6=I+I5
2410 J6=J+J5
2420 S3=0
2430 IF A(I6,J6)<>T2 THEN 25
```

```
90
2440 S3=S3+1
2450 I6=I6+I5
2460 J6=J6+J5
2470 IF A(I6,J6)=T1 THEN 250
0
2480 IF A(I6,J6)=0 THEN 2590
2490 GOTO 2440
2500 S1=S1+S3
2510 IF U<>1 THEN 2590
2520 I6=I
2530 J6=J
2540 FOR K1=0 TO S3
2550 A(I6,J6)=T1
2560 I6=I6+I5
2570 J6=J6+J5
2580 NEXT K1
2590 NEXT K
2600 RETURN
2610 FOR I=1 TO 8
2620 FOR J=1 TO 8
2630 CALL HCHAR(2*I+4,2*J+13
,ASC(D$(A(I,J)+1)))
2640 NEXT J
2650 NEXT I
2660 RETURN
2670 CALL SCREEN(4)
2680 CALL COLOR(14,16,1)
2685 DISPLAY AT(24,1):CTR$("
working, please wait")
2690 CALL CHAR(136,"08080808
FF0808080808080808080800000
0000FF000000")
2700 FOR R=5 TO 21 STEP 2 ::
 FOR C9=14 TO 30 STEP 2 :: C
ALL HCHAR(R,C9,136):: NEXT C
9 :: NEXT R
2710 FOR R1=6 TO 20 STEP 2 :
: FOR C8=14 TO 30 STEP 2 ::
CALL HCHAR(R1,C8,137):: NEXT
 C8 :: NEXT R1
2720 FOR R2=5 TO 21 STEP 2 :
: FOR C7=15 TO 29 STEP 2 ::
CALL HCHAR(R2,C7,138):: NEXT
 C7 :: NEXT R2
2730 FOR CH=65 TO 72 :: C3=C
H-59 :: CALL HCHAR(4,C3*2+3,
CH):: NEXT CH !CALL HCHAR(18
,C3+12,CH):: NEXT CH
2740 FOR CH=49 TO 56 :: C3=C
H-47 :: CALL HCHAR(C3*2+2,13
,CH):: NEXT CH !CALL HCHAR(C
3+8,26,CH):: NEXT CH
2750 RETURN
2760 END
```

■

TI990 MINICOMPUTER

by Tony McGovern, Australia

Yet another item spotted recently concerned the spiritual ancestor of the TI-99/4a, namely the 990 series minicomputers with their DX10/DNOS operating system. When you reflect on it, the minicomputer provenance is what has given the TI-99 expanded system it's remarkable staying power. The TMS-9900 processor was the 990 series architecture reduced to a single chip, and used in the low end of the 990 series. Other micros were based on grown-up calculator chips, and the influence is still felt in all those PCs based on Intel's flawed 8086-80286 architecture. Anyhow the sad news was that TI has just (Jun 1990) discontinued production of the 990 series, though support will continue to 1995. Sound familiar ? In this case the life cycle has pretty well been run through. All told 120,000 of various 990 minis were produced starting in 1971 and it is estimated that 5000 are still in active service. Can't say I have ever seen one, though I have seen plenty of PDP-11s and used DG Novas, which is the league the 990 was playing in. I suspect some of the earliest assembly programmers for the 99/4 learned their trade and even did some of their work on 990 minis. If you look at the opcode table in the 99/4 assembler, say by using <V>iew in DiskReview, you will see all sorts of things that do not exist on the 9900 including memory mapper instructions for the larger minis. This news finally does mark the passing of an era. ■

TI-BASE REFERENSER

PB 88-4 TI-Base 2.00 dec 1988
PB 89-4 TI-Base 2.02 aug 1989
PB 90-4 TI-Base 3.01 jun 1990
TI*MES 27 p.11: TI-Base tutorial
TI*MES 28 p. 9: TI-Base tips
Micropendium
maj 90: TI-Base upgraded to V3.0
jun 90: Speeding up loading times
jul 90: Speeding up operations
jul 90: TI-Base command file utility
aug 90: Selection of printer drivers
aug 90: TI-Base version 3.0 review ■

SWEDLOW EXTENDED BASIC

```
X   X   BBBB      # 15 to 16
 X X    B   B
  X     BBBB      By
 X X    B   B     Jim
X   X   BBBB      Swedlow
```

(This article originally appeared in the User Group of Orange County, California ROM)

RE-MAPPING THE KEYBOARD

You normally see CALL KEY(0,K,S). There are five other values for the first variable, the key unit. They remap your keyboard:

0 Keeps the keyboard in the same mode as the last time a CALL KEY was executed. If this is the key unit on the first CALL KEY in a program, you stay in 4A mode.

1 Splits the keyboard into two
2 smaller boards. Good for games.

3 Remaps the keyboard as a 99/4 (no A). CONTROL and lower case are inactive.

4 Pascal mode.

5 Remaps to the 99/4A mode with lower case and CONTROL active.

When you use 3, regardless of the position of SHIFT and ALPHA LOCK, all alphas return as upper case. The problem is that this condition continues until you do a CALL KEY with a different key unit. Try this:

```
10 CALL KEY(3,A,B) ::
   PRINT : :"Key Unit is 3"
20 INPUT A$ :: PRINT A$
30 CALL KEY(5,A,B) ::
   PRINT : :"Key Unit is 5"
40 INPUT A$ :: PRINT A$ ::
   GOTO 10
```

With ALPHA LOCK up, try inputting lower case letters and see what happens.

A key unit of 3 is very useful to make sure that only upper case alphas are caught by the CALL KEY. If you want lower case later in an INPUT or ACCEPT, however, you must remap the keyboard with another CALL KEY.

USER FRIENDLY/USER PROOF

When programming, you want your program to help the user. You also want to keep the user from crashing your program. Remember, the user will do most any fool thing. One area of vulnerability is inputting information thru INPUT and ACCEPT statements.

Lets say you want an integer between 1 and 9. Here are a number of ways you can input that number.

INPUT F can cause two problems. First, any number will be accepted. Second, if your user inputs anything but a number, you get: "WARNING: INPUT ERROR IN nnn TRY AGAIN".

This destroys your screen, scares your user and looks bad too.

ON WARNING can help. If you add ON WARNING NEXT the warning message will be suppressed but any number will be accepted. This coding is probably the best that can be done with INPUT F:

```
10 ON WARNING NEXT
20 INPUT F :: IF F<1 OR F>9 OR
   F<>INT(F) THEN 20
```

Bad values of F (0, 3.1, etc) will still cause the input prompt to be repeated and mess up your screen but you will get a good value in the end.

ACCEPT AT has a number of features that will help. With SIZE and VALIDATE you can avoid some problems:

```
10 ACCEPT AT(5,10)SIZE(1)
   VALIDATE(DIGIT)BEEP:F
```

Your user can still goof you up two ways. Zero is acceptable input and inputting a null will do strange things. SIZE(1) means no bigger than one character but it can be

smaller!

ACCEPT F$ will help solve some more problems. Strings will be accepted. This coding is about as user proof as you can get:

```
10 DISPLAY AT(5,1):"1"
20 ACCEPT AT(5,1)SIZE(-1)
   VALIDATE(DIGIT)BEEP:F$ ::
   IF F$="" THEN 20 ELSE
   F=MAX(VAL(F$),1)
```

By making the SIZE value negative, whatever is on the screen at (5.1) will be the default value if ENTER is pressed.

About the only way I have found to mess this up is to have a non-numeric sitting at (5,1). VALIDATE works ONLY on which key is pressed, it assumes that you know what is on the screen!

MEMORY SAVERS

This is the third and last article on how our TI's store stuff in memory. We have looked at the line number table and encoding line contents. This time we will draw some conclusions.

A variable name takes only its length in memory. <A> takes one byte while <MASTER@DEVICE> takes 13 bytes and <A$> takes two bytes.

A number, however, takes the number of characters plus two bytes. For example, the number 2.13 would take six bytes of memory and the number 1 would take three bytes.

Strings also take the number of characters in the string plus two. "He won" takes eight bytes.

Some suggestions:

-- Use the shortest possible variable and sub-program names.

-- If you use a given number often, assign it to a variable and use that instead.

Figure this way: A=2.13 takes eight bytes (one for A, one for = and six for 2.13). Each time you use A instead of 2.13 you save five bytes. Therefore, after two substitutions you are conserving memory.

-- Look for places where you can replace numerics with variables. This line:

```
10 C=0 :: INPUT A(0) :: PRINT A(0)
```

Would take four less bytes if done this way:

```
10 C=0 :: INPUT A(C) :: PRINT A(C)
```

Be sure, however, that you only do this when the variable must be the number you intent it to be!!!!!

SPEED

Longer variable names slow program execution. I ran this program with progressively longer variable names substituted for <C>:

```
10 C=0 :: FOR I=1 TO 1000 ::
   A=C :: NEXT I
```

I ran each three times and averaged the results. Here is what I found:

LENGTH OF SUB-STITUTE FOR <C>	AVERAGE RUN TIME (seconds)
1	7.74
2	7.72
3	7.76
4	7.90
5	8.04
6	8.12
7	8.23
8	8.34
9	8.46
10	8.56
11	8.62
12	8.71
13	8.83
14	8.94
15	9.02

As you can see, variables up to three characters in length ran in about the same time. Once the length was longer, however, each additional character in the variable name increased the run time by about one tenth of a second for 1000 exe-

cutions or .1 millisecond for one.

I also ran this one:

```
10 C=0 :: FOR I=1 TO 1000 ::
   A=0 :: NEXT I
```

The average run time was 7.06 seconds. There is a cost when substituting variables for numbers. TANSTAAFL (there ain't no such thing as a free lunch)!!

DID YOU KNOW?

You can have more than one item for a VALIDATE. For example,

```
10 ACCEPT AT(5,1)VALIDATE(DIGIT,"Q")
   BEEP:F$
```

Will accept the ten digits and the letter Q. It does.

ON BACK-UPS AND FLIPPIES

If you already back-up your disks faithfully and have decided about flippies, skip this. Otherwise, read on.

BACK-UP's are essential. The first thing you do when you get a new program or disk is back it up. Don't run it, don't modify it, don't catalog it, don't list it - back it up.

Why? Simply put, disks go bad and disk drives eat disks. If your only copy goes bad, it could take days to weeks to get a replacement, depending on the source. Ever read the 'warranty' that comes with some software?

So, make your back-up first. If you buy a program, keep the master (with the maker's label) with your back-ups and use a working copy for every day.

Keep your back-ups and masters in a separate disk box away from your computer. That way you won't use them by mistake.

If a disk does go bad, make sure that your hardware is working by using another disk BEFORE using your master or back-up. Otherwise, you could destroy both copies!

Updating back-ups is vital. If you wrote the program, you probably will revise it more than once. If you get a single program, you will normally add it to an existing disk. If you don't update your back-up, you will lose your new program if something goes wrong with your working disk.

A FLIPPY is a single sided disk that has been modified to act like two single sided disks. By adding three holes, you can put your disk in your drive upside down and record on the back.

Some folks, like Craig Miller, recommend against flippies. They argue that a disk is designed to turn one way and bad things happen when you flip it over and make it turn the opposite direction.

Others use flippies and claim that they have had no problems. One approach is to use the front as a working copy of one disk and the back as the back-up of another disk.

I compromise by only using flippies for back-up copies, thus reducing the number of back-up disks I need. I don't, however, use them for working disks. I haven't had any problems.

DISK COPIERS

There are two types of disk copiers: file-by-file and sector-by-sector. You need both.

FILE-BY-FILE copiers read your disk one file at a time and then write that file to the new disk. DISK MANAGER II, the module that came with the TI disk controller card, is a file copier that takes about three weeks to back-up a disk.

A much better file copier is DISK MANAGER 1000. This freeware item includes a sector copier and other features. It makes DISK MANAGER II obsolete.

SECTOR-BY-SECTOR copiers copy your disk without regard to the file

content. Instead, they read the master sector by sector and then write that information to the new disk, sector by sector.

Some sector copiers allow you to use or ignore the bit map. This is a table in sector zero that tells the disk controller which sectors are used and which are free. Using the bit map shortens copy time.

There are many good sector copiers. Freeware items include DISK MANAGER 1000 and MSCOPY. You can buy NIBBLER and TURBO COPY. TURBO COPY is the fastest copier I have found.

WHY BOTH? While sector copiers are faster, file copiers are more versatile and can repair some disk problems.

If the bit map is bad, copying the disk with a file copier will result in a good bit map on the new disk.

Another problem is a result of the way the disk controller saves files on a disk. Say that you are writing a letter and save it to disk three times: after a third is written, after two-thirds and when you are done. Each time the file will be larger.

If there is not enough space to write the new, larger file where the old, smaller file had been, the disk controller will save what it can where the file had been and the rest on another part of the disk.

The result is what Craig Miller calls a 'fractured file'. It takes longer to read a fractured file because the disk drive head must jump around more. This increases the probability of a read failure and of harm to your disk.

Millers Graphics' ADVANCED DIAGNOSTICS will tell you if you have fractured files. If you have done a lot of saving and changing on a disk, the odds are that you do.

The only way to unfracture a file is to copy your disk with a file copier.

Send freeware authors the price they request. Freeware is the best source of new TI products right now. It will dry up, however, without our support.

ASC AND SEG$

A common coding for determining the ASCII value of the first character of a string is:

```
ASC(SEG$(A$,1,1))
```

This can be simplified by omitting the SEG$ function:

```
ASC(A$)
```

ASC always returns the ASCII value of the first character of the string. You will get an error if A$ is a null string (if A$="").

Suppose you want to lop off the first four characters of a string. You might do this:

```
A$=SEG$(A$,5,LEN(A$)-4)
```

Our 4A's, however, do not compare the third value in the SEG$ function (the length of the new string) to the length of the old string. Therefore, this works just as well:

```
A$=SEG$(A$,5,255)
```

Use 255 as that is the maximum length of a string variable. If the length of A$ is less than 5 - even if it is zero the new A$ will be a null string (but NO error).

POSition

One way to ask for a menu selection is to ask the user to input the first letter of his/her choice. Suppose that the options were <C>hange, <P>rint or <Q>uit. You might do this:

```
190 ACCEPT AT(10,10)SIZE(-1)
    VALIDATE("CPQ"):A$ ::
    IF A$="C" THEN 230 ELSE
    IF A$="P" THEN 340 ELSE
    IF A$="Q" THEN 980 ELSE 190
```

A simpler way would be to use POS:

```
190 ACCEPT AT(10,10)SIZE(-1)
    VALIDATE("CPQ"):A$ ::
    ON POS("CPQ",A$,1) GOTO
    230,340,960
```

If your user inputs a null, the POS function will return a value of 1 and control will transfer to the first line in the GOTO list. If this is a problem, do it this way:

```
190 ACCEPT AT(10,10)SIZE(-1)
    VALIDATE("CPQ"):A$ ::
    IF A$="" THEN 190 ELSE
    ON POS("CPQ",A$,1) GOTO
    230,340,960
```

(Source: a Tigercub program)

DIM's and SUBPROGRAM's

Will this program work?

```
10 DIM A(5)
20 CALL SETUP(A())
30 SUB SETUP(B())
40 FOR I=1 TO 10
50 B(I)=I
60 NEXT I
70 SUBEND
```

Answer: It will crash in line 50 when I=6. Once A has been DIMentioned in line 10, the process of calling SETUP in line 30 transfers the DIMention to B within SETUP. In essence, there has been a DIM B(5) inside SETUP.

Note that a STOP is not needed at the end of line 20. Your 4A will not execute a SUBPROGRAM unless it is CALLed. ■

KÖPES XB-MANUAL
1 st Owners Manual till TI Extended Basic. L.A.Jones Tel: 031-53 21 70

KÖPES EXPANSIONSBOX
Köpes expansionsbox till TI-99/4A och PHP 1240 (TI diskkontrollkort) passande till ovanstående.
Erland Ericsson, Harestad
Tel. 0303-21 422

TEACH YOURSELF XB
Teach Yourself Extended Basic är en skiva med sju program för självstudier. Den är ett självständigt komplement till XB-manualen och har ursprungligen sålts av TI. En kopia kan fås genom att sända skiva+porto till Jan Alexandersson, Springarvägen 5, 3 tr, 142 61 TRÅNGSUND. ■

MULTIPLAN SYLK FILES - 1

by Bill Harms, USA

In this article I will introduce you to a method to transfer data from a basic program to Multiplan.

I use Multiplan to keep my budget and to estimate my income taxes. I have a spreadsheet with 18 columns: 12 months, Yearly Total, Year-To-Date, Weekly Average, Monthly Average and two for Taxes. Those last two have formulas to get various numbers from the spreadsheet. The rows include: Pay, Interest, Expenses, Loans and Other. You can really do "What If'ing" and "Why Not'ing" with Multiplan.

It was a bear, however, adding up all the darn monthly checkbook entries in separate categories (Meals Out, My Pay, My Wife's Pay, Groceries, Interest, etc).

Since Multiplan doesn't have a FOR/NEXT capability and cannot scan data until some criteria is met it was too time consuming to enter the 100 to 150 monthly entries directly from my checkbook into Multiplan (if there was enough RAM).

Now I use a nice little fast (I mean fast) Extended Basic program I wrote to get those Transactions added by category. Then I can use a SYLK creator to quickly and correctly prepare them for loading into my Multiplan spreadsheet.

SYLK (or Symbolic Link) files are a little known feature of Multiplan. They can be written to disk by a

basic program and read by Multiplan.

In this article I will show you how this is done. This material is based on a program I got from TI, a series of articles in the May (and later) 1985 SUPER 99 MONTHLY (now called THE SMART PROGRAMMER) and the Multiplan manual.

This bare bones program is based on the one I received from TI in 1984. The disclaimer was bigger than the program! It writes a disk file with a one cell spreadsheet that can be read by Multiplan.

```
100 OPEN #1:"DSK1.SYLKF",DISPLAY,
    OUTPUT,FIXED 128
110 CALL CLEAR
120 INPUT "ROW NUMBER: ";R$
130 INPUT "COLUMN NUMBER: ";C$
140 INPUT "CELL CONTENT: ";A$
150 FOR Q=1 TO 27-LEN(A$)
160 W$=W$&CHR$(0)
170 NEXT Q
180 X$=CHR$(34)&A$&CHR$(34) !
    Surrounds contents with quotes
190 Z$=CHR$(13)&CHR$(10) !
    Carrage Return and Line Feed
200 Y$="D;PMP"&Z$&"F;DG0GB"&
    Z$&"B;Y"&R$&";X"&C$&Z$&
    "C;K"&X$&Z$&"W;N1;A1 1"&
    Z$&"E"&Z$&W$
```

This monster of a line has the symbolics needed for Multiplan to read the file. See page 205 of the Multiplan manual for explanations.

```
210 PRINT #1:Y$
220 CLOSE #1
230 END
```

If you enter and run this program, you will find a file on your disk called 'SYLKF'. Before you can load this file, you must change it. It may seem a bit odd, but the file must be written as DISPLAY, FIXED 128 and then changed to INTERNAL, FIXED 128 in the file header. In other words, the file must use DISPLAY notation but must look like an INTERNAL file! There are two ways to do this. You will find an application of Barry Traver's RAW (Read And Write) in SUPER 99 MONTHLY. Or you can use ADVANCED DIAGNOSTICS to change the last four hex characters of the first line of the file header to <0202>.

Once you have done this you can load your file. First, boot Multiplan. Press <T>ransfer and then <O>ptions. Next press <S>ymbolic and then <ENTER>. Now press <T>ransfer again and this time <L>oad your file.

Here is the Multiplan spreadsheet:

```
|---------|
|       1 |
| 1 HARMS |
|         |
|---------|
```

This is what the data looks like on disk using Millers Graphics great ADVANCED DIAGNOSTICS:

```
Drive : 2   Track  : 3
Side  : 1   Sector : 34
Byte  : 0   Display: Ascii

I D ; P M P * * F ; D G O G
B * * B ; Y 1 ; X 1 * * C ;
K " H A R M S " * * W ; N 1
; A 1   1 * * E * * * * * *
```

Most of the **'s stand for CR/LF (Z$ in line 190).

There are many ways you could input data besides the simple INPUT in line 140. You could read data in from DATA statements or from a disk file. That disk file could be created by most anything: TI Writer, RS232, another module or a Multiplan Print File.

You can create data in Basic and then "dump" it into a spreadsheet en masse' instead of just keyboarding it.

You could transmit the outputted SYLK file of your Multiplan spreadsheet to others via RS232. The DIF (Data interchange Format) used by Lotus 1-2-3 and Visicalc only accommodates the cell content, not the sheet paramaters.

This is only a taste (BAD?) of what you can create to load data into Multiplan. It really opens Multiplan up to other software.

Next time we'll do a 2 row by 2 column spreadsheet. ■

TIPS FROM TIGERCUB #58.1

TIPS FROM THE TIGERCUB

#58.1

Tigercub Software
156 Collingwood Ave.
Columbus OH 43213

I am still offering over 120 original programs at $1 each, or on collection disks at $5 each. The five Tips From The Tigercub disks are reduced to $5 and the three Nuts & Bolts disks are now just $10 each.

My catalog is available for $1, deductable from your first order (specify TIGERCUB catalog).

****

TI-PD LIBRARY

I have selected public domain programs, by category, to fill over 300 disks, as full as possible if I had enough programs of the category, with all the Basic-only programs converted to XBasic, with an E/A loader provided for assembly programs if possible, instructions added and any obvious bugs corrected, and with an autoloader by full program name on each disk. These are available as a copying service for just $1.50 postpaid in U.S. and Canada. No fairware will be offered without the author's permission. Send SASE for list or $1, refundable for 11-page catalog listing all titles and authors. Be sure to specify TI-PD catalog.

****

In Tips #55 I published a CHARSUB routine to convert character patterns into assembly source code, and in Tips #55 and #56 I published several routines to manipulate hex codes into new character sets. Those patterns looked fine on my old TV, but when I demo'd them on a high-resolution monitor I could see too many missing pixels.

So I wrote this CHARFIX program which, when MERGEd into a program and CALLed after any character redefintion is completed, will permit any normal or reidentified character to be viewed on screen and edited and will then write the hex codes of any range of printable characters into an assembly source file which can be assembled, loaded and linked to instantly change character sets.

This routine also reidentifies the common punctuation into the same character sets as the letters, as described in Tips #55. If you do not want this feature, delete lines 29001-29003.

```
29000 SUB CHARFIX
29001 DATA 32,33,34,44,46
29002 RESTORE 29001 :: FOR J
=1 TO 5 :: READ CH :: CALL C
HARPAT(CH,CH$):: CALL CHAR(J
+90,CH$):: CALL CHAR(J+122,C
H$):: NEXT J
29003 CALL CHARPAT(63,CH$)::
 CALL CHAR(64,CH$):: CALL CH
AR(96,CH$)
29004 DISPLAY AT(1,1)ERASE A
LL:"1 2 3 4 5 6 7 8 9 0 : ;"
:" ":"@ A B C D E F G H I J
K L M":" ":"N O P Q R S T U
V W X Y Z [":" ":"\ ] ^ _ a
b c d e f g h i j"
29005 DISPLAY AT(9,1):"k l m
 n o p q r s t u v w x":" ":
"y z { | } ~"
29006 CALL CHAR(128,"FF"&RPT
$("81",6)&RPT$("FF",9)&"FFFF
"&RPT$("C3",4)&"FFFF"):: CAL
L COLOR(13,2,16)
29007 CALL CHARVIEW
29008 SUBEND
29009 SUB CHARVIEW
29010 DISPLAY AT(13,14):"CTR
L V TO VIEW" :: DISPLAY AT(1
```

```
4,14):" " :: DISPLAY AT(15,1
4):"CTRL E TO EDIT" :: DISPL
AY AT(17,14):"CTRL S TO SAVE
"
29011 DISPLAY AT(19,14):" "
:: DISPLAY AT(20,14):" "
29012 CALL KEY(0,@,S):: IF S
=0 THEN 29012 ELSE IF @=150
THEN 29015 ELSE IF @=133 THE
N 29014 ELSE IF @=147 THEN 2
9013 ELSE 29012
29013 CALL DELSPRITE(#1):: C
ALL CHARSUB(HX$()):: DISPLAY
 BEEP :: STOP
29014 CALL EDIT(K):: GOTO 29
010
29015 DISPLAY AT(24,1)BEEP:"
"
29016 DISPLAY AT(24,1):"PRES
S A KEY" :: CALL KEY(0,K,S):
: IF S<1 OR K<32 OR K>143 TH
EN 29016
29017 DISPLAY AT(24,1):"" ::
 CALL CHARPAT(K,CH$)
29018 R=13 :: FOR J=1 TO 15
STEP 2
29019 H$=SEG$(CH$,J,1):: CAL
L HEX_BIN(H$,B$)
29020 H$=SEG$(CH$,J+1,1):: C
ALL HEX_BIN(H$,BB$):: FOR L=
1 TO 8 :: C$=C$&CHR$(ASC(SEG
$(B$&BB$,L,1))+80):: NEXT L
29021 DISPLAY AT(R,1):C$;::
DISPLAY AT(R,10):SEG$(CH$,J,
2);:: R=R+1 :: C$="" :: NEXT
 J :: DISPLAY AT(22,1):CH$;:
: GOTO 29012
29022 SUBEND
29023 SUB HEX_BIN(H$,B$):: H
X$="0123456789ABCDEF" :: BN$
="0000X0001X0010X0011X0100X0
101X0110X0111X1000X1001X1010
X1011X1100X1101X1110X1111"
29024 FOR J=LEN(H$)TO 1 STEP
 -1 :: X$=SEG$(H$,J,1)
29025 X=POS(HX$,X$,1)-1 :: T
$=SEG$(BN$,X*5+1,4)&T$ :: NE
XT J :: B$=T$ :: T$="" :: SU
BEND
29026 SUB CHARSUB(HX$())
29027 DISPLAY AT(12,1)ERASE
ALL:"Source code filename?":
"DSK" :: ACCEPT AT(13,4)SIZE
(12)BEEP:F$ :: OPEN #1:"DSK"
&F$,OUTPUT
29028 DISPLAY AT(15,1):"LINK
ABLE program name?" :: ACCEP
T AT(16,1)SIZE(6):P$
29029 DISPLAY AT(18,1):"Rede
fine characters from    ASCI
I     to ASCII"
29030 ACCEPT AT(19,7)VALIDAT
E(DIGIT)SIZE(3):F
29031 ACCEPT AT(19,21)VALIDA
TE(DIGIT)SIZE(3):T
29032 PRINT #1:TAB(8);"DEF";
TAB(13);P$ :: PRINT #1:"VMBW
   EQU  >2024" :: PRINT #1:"
STATUS EQU  >837C"
29033 NB=(T-F)*8 :: CALL DEC
_HEX(NB,H$):: A=768+F*8 :: C
ALL DEC_HEX(A,A$)
29034 FOR CH=F TO T :: IF CH
<144 THEN CALL CHARPAT(CH,CH
$)ELSE CH$=HX$(CH)
29035 IF FLAG=0 THEN PRINT #
1:"FONT";:: FLAG=1
29036 FOR J=1 TO 13 STEP 4 :
: M$=M$&">"&SEG$(CH$,J,4)&",
" :: NEXT J :: M$=SEG$(M$,1,
23)&"  *"&CHR$(CH)
29037 PRINT #1:TAB(8);"DATA
"&M$ :: M$="" :: NEXT CH
29038 PRINT #1:P$;TAB(8);"LI
   R1,FONT" :: PRINT #1:TAB(
8);"LI   R0,>"&A$ :: PRINT #
1:TAB(8);"LI   R2,>"&H$
29039 PRINT #1:TAB(8);"BLWP
@VMBW":TAB(8);"CLR  @STATUS"
:TAB(8);"RT":TAB(8);"END" ::
 CLOSE #1
29040 SUBEND
29041 SUB DEC_HEX(D,H$)
29042 X$="0123456789ABCDEF"
:: A=D+65536*(D>32767)
29043 H$=SEG$(X$,(INT(A/4096
)AND 15)+1,1)&SEG$(X$,(INT(A
/256)AND 15)+1,1)&SEG$(X$,(I
NT(A/16)AND 15)+1,1)&SEG$(X$
,(A AND 15)+1,1):: SUBEND
29044 SUB EDIT(CH)
29045 DISPLAY AT(13,14):"1 T
O TOGGLE" :: DISPLAY AT(14,1
5):"CURSOR" :: DISPLAY AT(15
,14):"E S D X TO MOVE" :: DI
SPLAY AT(17,14):"CTRL A TO A
BORT"
29046 DISPLAY AT(19,14):"CTR
L R TO" :: DISPLAY AT(20,15)
:"REIDENTIFY"
29047 R=13 :: C=3 :: X=128 :
: CALL SPRITE(#1,130,11,R*8-
7,C*8-7):: X$=CHR$(129)&CHR$
(146)
29048 CALL KEY(0,K,S):: IF S
<1 THEN 29048 ELSE ON POS("1
EeSsDdXx"&X$,CHR$(K),1)+1 GO
TO 29048,29049,29050,29050,2
9051,29051,29052,29052,29053
,29053,29055,29056
29049 X=X+1+(X=129)*2 :: GOT
O 29054
```

```
29050 R=R-1-(R=13):: GOTO 29
054
29051 C=C-1-(C=3):: GOTO 290
54
29052 C=C+1+(C=10):: GOTO 29
054
29053 R=R+1+(R=20)
29054 CALL LOCATE(#1,R*8-7,C
*8-7):: CALL HCHAR(R,C,X)::
GOTO 29048
29055 CALL DELSPRITE(#1):: S
UBEXIT
29056 FOR R=13 TO 20 :: FOR
C=3 TO 10 :: CALL GCHAR(R,C,
GH):: CALL LOCATE(#1,R*8-7,C
*8-7):: B$=B$&CHR$(GH-80)::
NEXT C
29057 CALL BIN_HEX(B$,H$)::
DISPLAY AT(R,10):H$;:: B$=""
 :: HEX$=HEX$&H$ :: NEXT R :
: DISPLAY AT(22,1):HEX$;:: C
ALL CHAR(CH,HEX$):: HEX$=""
29058 CALL DELSPRITE(#1):: F
OR R=13 TO 20 :: DISPLAY AT(
R,14):"" :: NEXT R :: SUBEND
29059 SUB BIN_HEX(B$,H$):: H
X$="0123456789ABCDEF" :: BN$
="0000X0001X0010X0011X0100X0
101X0110X0111X1000X1001X1010
X1011X1100X1101X1110X1111"
29060 L=LEN(B$):: IF L/4<>IN
T(L/4)THEN B$="0"&B$ :: GOTO
 29060
29061 FOR J=L-3 TO 1 STEP -4
 :: X$=SEG$(B$,J,4)
29062 X=(POS(BN$,X$,1)-1)/5
:: T$=SEG$(HX$,X+1,1)&T$ ::
NEXT J :: H$=T$ :: T$="" ::
SUBEND
```

I think that programs, at least non-commercial ones, should be open for anyone to modify for their own use. For that reason, I would not normally publish the following routine. However, I recently received a large number of programs, originally in the IUG library, and found that the author's name had been erased from the title screen or REM of every one of them. I know, because I already had many of the original versions, including some that I wrote myself.

Now, that is inexcusable. If a programnmer is willing to share his work, he does deserve credit for it. And if people are going to play that dirty, maybe there is good reason for protecting programs.

So here is how to do it. Ken Woodcock wrote this ingenious routine and published it in the Tidewater newsletter. I have modified it so that it can be deleted after it has done its work. It is to be MERGEd into any XBasic program(32k required) and RUN, and will change the line length byte of each line to zero, so that the program cannot be LISTed, although it can be loaded and run.

```
1 CALL INIT :: CALL PEEK(-31
952,A,B,C,D):: SL=C*256+D-65
539 :: EL=A*256+B-65536 :: F
OR X=SL TO EL STEP -4
2 CALL PEEK(X,E,F,G,H):: ADD
=G*256+H-65536 :: J=J+1 :: I
F J<4 THEN 3 :: CALL LOAD(AD
D-1,0)
3 NEXT X :: STOP :: !@P-
```

Save that as FIX in MERGE format. Merge it into any program (RESequence first if it has line numbers less than 4) and RUN. Then type 1, FCTN X and FCTN 3 to delete line 1. Delete lines 2 and 3 in the same way. Then SAVE. Now try LISTing it and watch the fireworks.

Ken wrote an even more ingenious UNFIX routine to unprotect the program, but I'm not passing that on!

Now, suppose you have a party game program that you don't want the kids playing with. So, RESequence it to some odd number, such as RES 797. Put in a line just before that 796 STOP . Then merge in FIX, run it, and delete those first 3 lines.

I hope you remember what line number you resequenced it to start from, because now you can only run it by RUN 797 !

In Tips #57 I reported the

discovery that printing to the disk from the TI- Writer Formatter, with the C option, really converted the carriage returns to trailing blank ASCII 32's, and I published a routine to strip them. I have found an easier way. First PF and C DSK... to convert the CRs to blanks. LF DSK... and SF DSK... to strip out those blanks, but that leaves the pestiferous tab line, so LF DSK... and PF DSK... again!

The first few disks of Tips #58 that I sent out had a poor version of this program. This is the corrected version. First key this in -

```
1 DISPLAY AT(12,1)ERASE ALL:
"SKIP INSTRUCTIONS? Y" :: AC
CEPT AT(12,20)SIZE(-1)VALIDA
TE("YNyn"):@Q$ :: IF @Q$="Y"
 OR @Q$="y" THEN 8
2 DISPLAY AT(24,5)ERASE ALL:
"PRESS ANY KEY"
3 RESTORE 30721
4 REM
5 FOR J@=1 TO T@ :: READ @$
:: DISPLAY AT(J@,1):@$:" "
6 CALL KEY(0,K@,S@):: IF S@=
0 THEN 6
7 NEXT J@
8 DATA 0
9 RESTORE 8 :: READ N
10 REM
```

Save it by -
SAVE DSK1.D/MERGE,MERGE
Then key this in -

```
100 OPEN #1:"DSK1.D/MERGE",V
ARIABLE 163,INPUT :: OPEN #2
:"DSK1.D/MERGE2",VARIABLE 16
3,OUTPUT :: L=129 :: FOR J=1
 TO 10
110 LINPUT #1:M$ :: PRINT #2
:CHR$(0)&CHR$(L+J)&CHR$(156)
&CHR$(253)&CHR$(200)&CHR$(1)
&"1"&CHR$(181)&CHR$(199)&CHR
$(LEN(M$))&M$&CHR$(0):: NEXT
 J
120 CLOSE #1 :: PRINT #2:CHR
$(255)&CHR$(255):: CLOSE #2
```

Run it to convert D/MERGE into a merge format file D/MERGE2 on DSK1. Then key this in. Don't change line numbers.

```
100 CALL CLEAR :: OPEN #1:"D
SK1.@DATA",VARIABLE 163,OUTP
UT :: DEF L$(X)=CHR$(120)&CH
R$(X)
105 PRINT #1:L$(X)&CHR$(161)
&CHR$(200)&CHR$(6)&"@DUMMY"&
CHR$(0)
110 L=L+1 :: X=X+1 :: ACCEPT
 AT(L,0):M$ :: IF L=24 THEN
CALL CLEAR :: L=0
120 IF M$<>"END" AND M$<>"en
d" THEN PRINT #1:L$(X)&CHR$(
147)&CHR$(199)&CHR$(LEN(M$))
&M$&CHR$(0):: GOTO 110
130 REM
140 PRINT #1:CHR$(0)&CHR$(4)
&"T@"&CHR$(190)&CHR$(200)&CH
R$(LEN(STR$(X-1)))&STR$(X-1)
&CHR$(0)
141 PRINT #1:L$(X)&CHR$(168)
&CHR$(0)
150 PRINT #1:CHR$(255)&CHR$(
255):: CLOSE #1
```

Enter MERGE DSK1.D/MERGE2 to merge in that file. SAVE the program as DATAWRITER. Then RUN it and try it out by using it to write itself some instructions. Answer the prompts with -

DATAWRITER V1.2
by Jim Peterson

To be used to add instructions to programs.

Type the instructions and format them, centered or hyphenated or right-adjusted just as you want them to appear on screen, and enter each line. They will be written to a D/V163 file named @DATA. When finished, enter END.

Then enter NEW, then MERGE DSK1.@DATA, and RUN to see if everything is OK. If so, load the program needing instructions, make sure its lowest line number is more than 10 and the highest is less than 30721, and enter MERGE DSK1.@DATA.

And enter END, then OLD DSK1.DATAWRITER, then MERGE DSK1.@DATA. ■

PUTTING IT ALL TOGETHER

by Jim Peterson, Tigercub, USA

The hard part of learning to program is not in learning what the various commands do - it is in learning to put them together to do what you want them to do!

Key in this little program and run it to see what it does, then read the explanation of how it does it.

```
100 DISPLAY AT(3,11)ERASE AL
L:"SPELLIT" !by Jim Peterson
110 DATA HIPPOPOTAMUS,CRITIQ
UE,KHAKI,IRIDESCENT,ARCHAIC,
PNEUMONIA
120 !add as many DATA statem
ents as you want
130 FOR CH=97 TO 122 :: CALL
 CHARPAT(CH-32,CH$):: CALL C
HAR(CH,CH$):: NEXT CH :: CAL
L COLOR(9,8,2,10,8,2,11,8,2,
12,8,2)
140 DATA END
150 READ M$ :: T=100 :: IF M
$="END" THEN CALL CLEAR :: S
TOP
160 GOSUB 230 :: ACCEPT AT(1
2,1)SIZE(-28)BEEP:Q$
170 IF Q$=M$ THEN CALL SOUND
(100,392,5):: CALL SOUND(200
,523,5):: DISPLAY AT(12,1):"
" :: GOTO 150
180 FOR J=1 TO LEN(Q$):: IF
SEG$(Q$,J,1)=SEG$(M$,J,1)THE
N 210
190 DISPLAY AT(12,J):CHR$(AS
C(SEG$(Q$,J,1))+32);
200 T=T+50 :: IF LEN(Q$)=LEN
(M$)THEN GOSUB 230 :: GOTO 2
10 ELSE DISPLAY AT(12,J+1):"
 " :: J=LEN(Q$):: GOTO 160
210 NEXT J
220 T=T+50 :: GOTO 160
230 DISPLAY AT(10,1):M$ :: F
OR D=1 TO T :: NEXT D :: DIS
PLAY AT(10,1):"" :: RETURN
```

Line 100 erases all the trash from the screen and prints the title centered on line 3. The screen is 28 characters wide. SPELLIT contains 7 characters. 28 minus 7 divided by 2 is 10.5, so center the title at column 11.

Put in as many lines of words in DATA as you want.

The lower case characters "a" through "z" are ASCII 97 to 122. The upper case are just 32 below that, ASCII 65 to 90. CALL CHARPAT to get the hex pattern identifier of each upper case letter in CH$, then CALL CHAR to reidentify the corresponding lower case letter to that pattern. The lower case letters are in sets 9 to 12, so color them in the reverse of the normal screen, cyan on black.

The dummy data END in line 140, and the statement in line 150, causes the program to stop without crashing when it runs out of words, regardless of how many you put in.

Line 150 reads each word from DATA one after another and sets the initial time to display it at 100 milliseconds.

Line 160 jumps to line 230 to display the word at line 10 column 1, wait for the set time, then erase it by displaying a null string which erases the line. Then it signals with a beep and cursor that it is waiting for your spelling, Q$, at line 12 column 1.

Line 170 checks whether your spelling is the same as the word M$. If so, it sounds two notes, displays a null string to erase the word and goes back for the next.

If not correct, line 180 starts a loop for the number of letters, LEN(Q$), in your spelling and compares each letter with the letter in the same position in M$. If the same, it jumps to 210 to check the next letter.

But if incorrect, line 190 displays at that point the character of the ASCII 32 higher, which is the same letter in inverted colors.

Line 200 increments the flashing time by 50, then checks to see if the word you spelled is the same length as the correct word. If so, it goes to 230 to flash the correct word, then continues checking letters. When finished, line 220 increments flash time and sends you back to try again. The -28 size in the ACCEPT statement prevents the misspelled word from being erased.

If your spelling has a different number of letters, the first error probably has caused all subsequent letters to be in the wrong position. They would all be marked as wrong, so a null string is displayed to erase the rest of the word. The statement J=LEN(Q$) clears the loop in computer memory to avoid the possibility of a MEMORY FULL error. Then you are prompted to try spelling the word again. ■

SAMPLING AV LJUD - EA

av Lars Herold Andersen, Danmark

```
****                    ******************************
*                       *                            *
*                       * 'SAMPLER' update 15.Juli 89 *
*                       *   EDITOR/ASSEMBLER VERSION  *
*                       *                         lha *
*                       ******************************
*
*     Denne version af 'SAMPLER' er beregnet til at loades i LOW MEMORY-EXP.
*     Læs mere om programmet i source-filen MINIMEM/S i PB 90-1.
*
*
*     Kørsels-instruktioner for EXPANSION MEM versionen :
*
*              VÆLG EDIT/ASSM , LOAD & RUN
*              FILENAME:        DSKn.EXPMEM/O   ( Programmet auto-starter )
*              Sæt musik på kassette-afspilleren, sæt volume højt, og sørg for,
*              at kassette-kablet er tilsluttet. Følg nu instruktionerne på
*              skærmen.
*
****

       DEF  SAMPLR


USRWSP EQU  >8300        VÆLG ET KVIKT WORKSPACE
MAGIN  EQU  >1B          HER BOR KASSETTEKABLETS HVIDE LEDNING
AUDIO  EQU  >18          HER SLÅS LYD-UDGANGEN TIL OG FRA
STATUS EQU  >837C        ADRESSEN FOR GPL STATUS BYTE. UNDERSØGES EFTER KEYSCANS
ISR    EQU  >83C4        HER LOADES STARTADRESSEN FOR USER-DEFINED INTERRUPT
VSBW   EQU  >210C        UTILITY-VEKTORER FOR EDITOR/ASSEMBLER
VMBW   EQU  >2110
VWTR   EQU  >211C
KSCAN  EQU  >2108


       AORG >2676        FØRSTE FRIE ADRESSE I LOW MEMORY-EXPANSION

SAMPLR LWPI USRWSP

       LI   R0,>01F0     VÆLG TEXT MODE
       BLWP @VWTR
       SWPB R0
       MOVB R0,@>83D4    HUSK - EN KOPI AF VDP-reg 1 HERTIL !

       CLR  R12          SØRG FOR DEN RETTE CRU-BASE

       MOV  @LINK,@ISR   FORTÆL @>83C4 OM INTERRUPT-RUTINENS PLACERING

**
*
* 'SAMPLER' BENYTTER SIG AF TO SKÆRME;
* EN FRA OG MED >0000 TIL OG MED >03BF, OG
* EN FRA OG MED >0C00 TIL OG MED >0FBF.
*
**
```

```
       LI   R0,>3BF       SKÆRMEN SLETTES FRA BUNDEN; POSITION #959 FØRST
       LI   R1,>2000      ASCII-KODEN FOR >SPACE< ER >20
CLRLOP BLWP @VSBW         >SPACE< SKRIVES TIL VDP-ADRESSEN I R0 ( SKÆRM 1 )
       SOC  @ADDER,R0     LÆG >0C00 TIL R0
       BLWP @VSBW         >SPACE< SKRIVES TIL VDP-ADRESSEN I R0 ( SKÆRM 2 )
       SZC  @ADDER,R0     TRÆK >0C00 FRA R0
       DEC  R0
       JOC  CLRLOP        FYLD MED >SPACE< INDTIL R0 GÅR FRA >0000 TIL >FFFF
*                         ( ALTSÅ NÅR POSITION #0 OG #>C00 ER OVERSKREVET )

**
* LÆG TEKST PÅ DE TO SKÆRME
**

       LI   R0,>0030      HER SKAL TEKSTEN PÅ SKÆRM 1 STARTE
       LI   R1,TEXT1      HER FINDES DEN,
       LI   R2,103        SÅ MANGE BYTES FYLDER DEN,
       BLWP @VMBW         OG NU SKRIVES DEN...
       LI   R0,>0C32      HER SKAL TEKSTEN PÅ SKÆRM 1 STARTE
       LI   R1,TEXT2      O.S.V...
       LI   R2,20
       BLWP @VMBW

**
*           HER STARTER DET EGENTLIGE PROGRAM.
* PROGRAMMET RESTARTER OGSÅ HER EFTER ET BRUGER-INTERRUPT
**

RESTRT
       SBZ  AUDIO         ENABLE AUDIO-GATE ( KAN VÆRE RESAT I INTRPT-ØJEBLIKKET)
       LIMI 0             INTERRUPT FORBUDT
       LI   R0,>07FC      SKÆRMEN SKAL VÆRE GRØN ( C ) MED HVID FORGRUND ( F )
       BLWP @VWTR
       LI   R0,>0200      SKÆRM 1 SKAL BRUGES; DEN STARTER VED >0000, ALTSÅ
       BLWP @VWTR         >00 TIL VDP-reg 2


**
* PROGRAMMET HAR LAGT TEKST 1 PÅ EN GRØN SKÆRM,
*       OG NU MÅ BRUGEREN BESLUTTE SIG
**

       CLR  @>8374        DET ALMINDELIGE KEYBOARD SKAL SKANNES
SCAN   BLWP @KSCAN
       MOVB @STATUS,R4    KIG PÅ RESULTATET AF SKANNINGEN
       COC  @KEYBIT,R4    ER DER TRYKKET PÅ EN TAST ?
       JNE  SCAN          HVIS IKKE, SÅ SKANNES DER ENDNU ENGANG

**
* NÅR EN TAST ER RAMT, MÅ PROGRAMMET VIDERE
**

       LI   R0,>07F6      SKÆRMEN SKAL VÆRE MØRK-RØD ( 6 ), FORGRUNDEN HVID
       BLWP @VWTR
       LI   R0,>0203      SKÆRM 2 SKAL BRUGES; DEN STARTER VED 3 * >400 = >C00
       BLWP @VWTR         ALTSÅ >03 TIL VDP-reg 2


**
* PROGRAMMET HAR LAGT TEKST 2 PÅ EN RØD SKÆRM -
*     SKAL DER INDSPILLES ELLER AFSPILLES ?
**
```

```
       MOV  @>8374,R4    HENT ASCII-KODEN FOR DEN TAST SOM RAMTES
       COC  @ENTKEY,R4   BLEV DER TRYKKET PÅ >ENTER< ?
       JEQ  SAMPLE       HVIS JA, SÅ SKAL DER SAMPLES. ELLERS ER DET PLAYBACK..


*************************
***                   ***
**   PLAYBACK AFSNIT   **
*                       *
*                       *

PLYBCK
       MOV  @MIN1,R0     AFSPILNINGEN STARTER FRA MIN1
LOOP1  C    R0,@MAX1     ER R0 NÅET FREM TIL SLUTNINGEN AF LOW MEMORY ?
       JNE  $+6          HVIS NEJ, SÅ SKIP NÆSTE INSTRUKTION
       MOV  @MIN2,R0     HVIS JA, SÅ FORTSÆTTER AFSPILNINGEN FRA START AF HI-MEM
       C    R0,@MAX2     ER R0 NÅET FREM TIL SLUTNINGEN AF HIGH MEMORY ?
       JHE  PLYBCK       HVIS JA, SÅ PLAYBACK IGEN
       LI   R2,>F        DER ER 16 BITS I ET WORD. DET SKAL SHIFTES 15 GANGE
       MOV  *R0+,R1      HENT ET WORD IND I R1 FOR DISSEKTION. FORØG R0 MED 2
LOOP2  COC  @LSBIT,R1    ER BIT 15 SAT ?
       JNE  SILNC1       HVIS NEJ, SÅ ER DET TYST
       SBO  AUDIO        HVIS JA, SÅ BLIVER AUDIO-BIT'EN HØJ
       JMP  $+4          SPRING VIDERE
SILNC1 SBZ  AUDIO        AUDIO-BIT'EN BLIVER LAV


       LI   R3,1         EN LILLE FORSINKELSE, OM MAN VIL.
DELAY1 DEC  R3           JO HØJERE VÆRDI ( >= 1 ) I R3, DES LAVERE
       JNE  DELAY1       AFSPILNINGS-HASTIGHED


       SRL  R1,1         SHIFT R1 TIL HØJRE; KLAR TIL AT UNDERSØGE NÆSTE BIT
       DEC  R2           FORMINDSK SHIFT-TÆLLEREN
       JNE  LOOP2        HVIS IKKE ALLE BITS I WORD'ET ER AFSPILLET SÅ GØR DET
       LIMI 2            GIV BRUGEREN LIDT INDFLYDELSE; TILLAD VDP-INTERRUPTS
       LIMI 0            SLUT MED DET !
       JMP  LOOP1        VIDERE TIL NÆSTE WORD I RÆKKEN
*                       *
*                       *
**                     **
***                   ***
*************************


*************************
***                   ***
**   SAMPLER AFSNIT    **
*                       *
*                       *

SAMPLE
       MOV  @MIN1,R0     INDSPILNINGEN STARTER I MIN1
LOOP3  C    R0,@MAX1     ER R0 NÅET FREM TIL SLUTNINGEN AF LOW MEMORY ?
       JNE  $+6          HVIS NEJ, SÅ SKIP NÆSTE INSTRUKTION
       MOV  @MIN2,R0     HVIS JA, SÅ FORTSÆTTER INDSPILNINGEN I START AF HI-MEM
       C    R0,@MAX2     ER R0 NÅET FREM TIL SLUTNINGEN AF HIGH MEMORY ?
       JHE  RESTRT       HVIS JA, ER INDSPILNINGEN SLUT; TILBAGE TIL RESTART
       LI   R2,>F        ELLERS INITIALISERES SHIFT-TÆLLEREN
LOOP4  SRL  R1,1         SHIFT TIL HØJRE; BIT 0 ER KLAR TIL AT MODTAGE EN SAMPLE
```

```
       LI   R3,1          JO HØJERE VÆRDI ( >= 1 ) I R3, DES LAVERE SAMPLING-RATE
DELAY2 DEC  R3
       JNE  DELAY2


       TB   MAGIN         ER DEN HVIDE KASSETTELEDNING LOGISK HØJ ?
       JNE  SILNC2        HVIS IKKE ER DET TYST; BIT 0 FORBLIVER 0
       SOC  @MSBIT,R1     INDGANGEN VAR HØJ; BIT 0 BLIVER SAT
SILNC2 DEC  R2            FORMINDSK SHIFT-TÆLLEREN
       JOC  LOOP4         HVIS IKKE ALLE BITS I WORD'ET ER INDSPILLET, SÅ GØR DET
       MOV  R1,*R0+       FLYT DET INDSPILLEDE WORD UD I MEMORY
       LIMI 2             GIV BRUGEREN LIDT INDFLYDELSE; TILLAD VDP-INTERRUPTS
       LIMI 0             SLUT MED DET !
       JMP  LOOP3         VIDERE TIL NÆSTE WORD I RÆKKEN
*                        *
*                        *
**                      **
***                    ***
************************


**
* HER FINDES INTERRUPT-RUTINEN, SOM UDFØRES 50 GANGE I SEKUNDET
**


INTRPT
       LWPI USRWSP        LOAD EGET WORKSPACE. INTERRUPT-RUTINEN HAR VALGT GPLWS
       BLWP @KSCAN        KEYBOARD'ET SKANNES...
       MOVB @STATUS,R4
       COC  @KEYBIT,R4
       JNE  $+8           HVIS INGEN TAST ER RAMT, SÅ SKIP NÆSTE INSTRUKTION
       MOV  @REST,@>83DC HVIS EN TAST ER RAMT, SKAL DER RETURNERES TIL RESTRT.
*                         ROM-INTERRUPT-RUTINEN RETURNERER FRA INTWS, SÅ VED AT
*                         LÆGGE ADRESSE FOR RESTRT I INTWS' R14 ( @>83DC ) BÆRES
*                         PROGRAM-COUNTEREN TIL DET RETTE STED AF RTWP !
       LWPI >83E0         RESTORE GPLWS
       RT                 RETURNER


**
* NOGLE NYTTIGE DATA
**

MIN1   DATA >27BA         FØRSTE FRIE ADRESSE I LOW MEMORY
MAX1   DATA >3FFE         SIDSTE FRIE ADRESSE I LOW MEMORY
MIN2   DATA >A000         FØRSTE FRIE ADRESSE I HIGH MEMORY
MAX2   DATA >FFE0         SIDSTE FRIE ADRESSE I HIGH MEMORY
ADDER  DATA >0C00         BRUGES VED SLETNING AF SKÆRMENE
ENTKEY DATA >000D         ASCII-KODEN FOR >ENTER< I MSBYTE
KEYBIT DATA >2000         BRUGES VED KSCAN. MASKE FOR BIT 2
MSBIT  DATA >8000         BIT 0
LSBIT  DATA >0001         BIT 15
LINK   DATA INTRPT        HER FINDES BRUGER-INTERRUPTETS START-ADRESSE
REST   DATA RESTRT        HER FINDES RESTART-RUTINENS START-ADRESSE
TEXT1  TEXT 'PRESS >ENTER< TO SAMPLE'
       TEXT '                          '
       TEXT '                         '
       TEXT '-ANY OTHER TO PLAY BACK'
TEXT2  TEXT 'HIT ANY KEY TO ABORT'

       END  SAMPLR                                                    ■
```