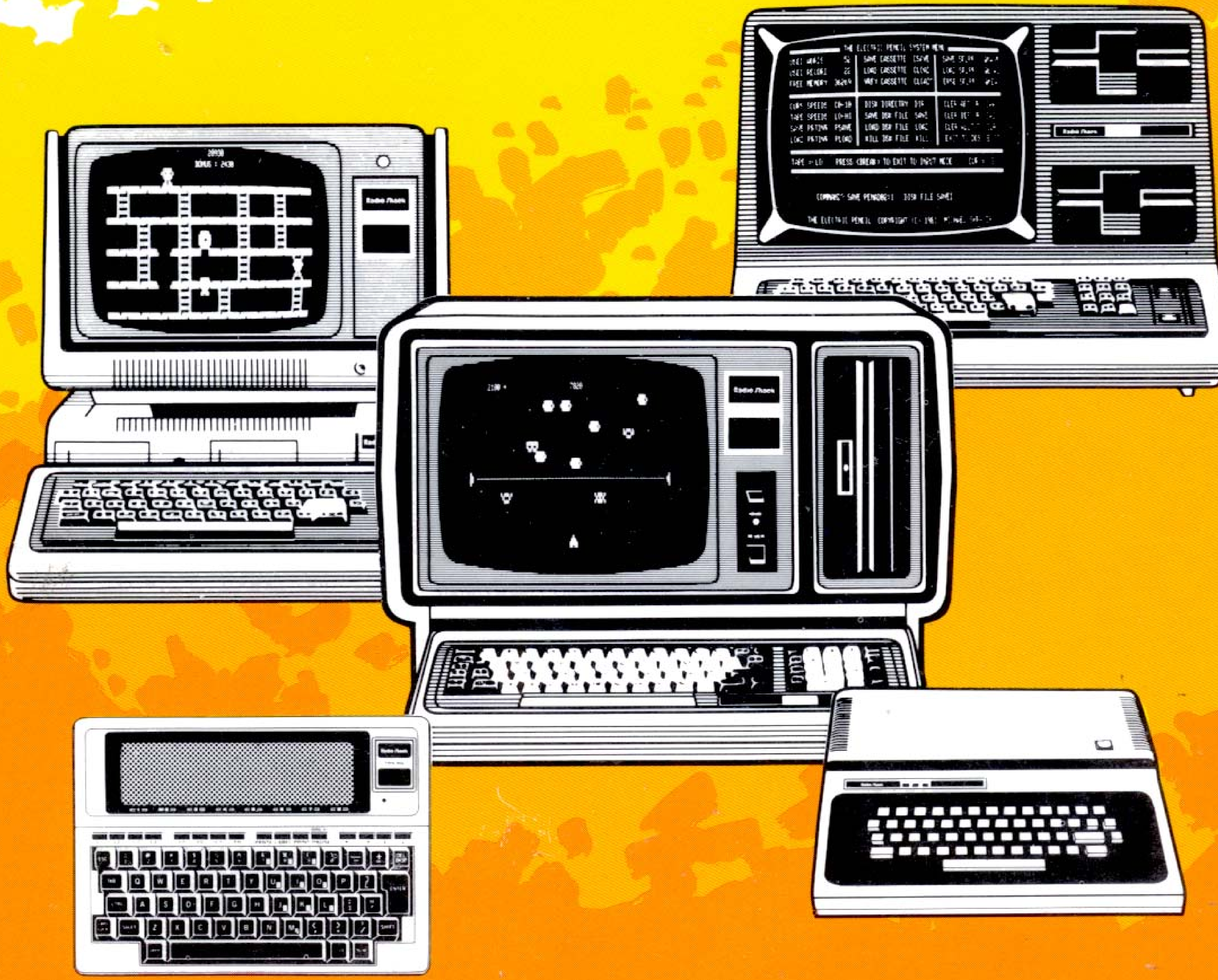
William Barden, Jr.

HOW TO DO IT on the TRS-80®

for the

Model I, II, III, Color Computer, and Model 100



William Barden, Jr.

HOW TO DO IT on the TRS-80®

for the
Model I, II, III, Color Computer, and Model 100

Editor-in-Chief: David Moore — *Spiritual Direction*
Managing Editor: Charles Trapp — *Gunslinger*
Cover Design and Graphics: D. J. Smith — *Bass Guitar & Vocals*
Technical Editor: Paul Wiener — *Unique humor*
Production & "Re-production": Cindy Hall — *Energy Supply*

First Edition
First Printing April 1983
Printed in the United States of America
Copyright©1983 by IJG Inc.

ISBN 0 936200 08 1

Published by
IJG Inc
1953 West
11th Street
Upland,CA
91786 (714)
946-5805

IMPORTANT

Read This Notice

Preface

Why are computers so hard to use? I remember spending the better part of a day trying to figure out what keyboard character was required to terminate a line of input. The character wasn't detailed in the manufacturer's documentation, nobody who had worked with the machine knew, and by gosh, nobody at the factory knew when I called them, either. (Sorry about the gosh, but we're trying to appeal to middle America with this book).

The incident above took place on a minicomputer system, but things haven't changed much since then. I own a Radio Shack TRS-80 Model I, Model II, Model III,and Color Computer, among others, and I am constantly running into small problems in operation, hardware questions, and software problems. Oh sure, some of the problems are documented, but many are not, or they're buried in the depths of some incomprehensible manual.

How about you? Have you ever wanted to set the RS-232-C switches on your Model I, but couldn't remember the sequence? Did you ever want to speed up and compact your BASIC code, but couldn't remember what to do? Have you ever wanted to connect the Color Computer and a modem, but didn't really know how? Did you ever want to hook up your Model I or III to a burglar alarm, but just didn't know how to implement it? Did you ever see a great program in assembly language in 80-MACRO magazine, but couldn't figure out how to use it on your 32K system? Did you ever want to solder a printer cable connection, but didn't know how to approach it? *Is that what's troubling you, bub*?

They're here. The answers. You can use this book and it'll tell you how to do all of the above and much, much, more. I've tried to write the book to provide as many answers as possible to common problems: hardware, software, and procedural, on the Model I, II, III,and Color Computer.

Furthermore, I've tried to make the answers idiot-proof. Now don't get riled up by that term (y'hear?). I'm one of computing's biggest idiots; I'm continually forgetting common, step-by-step procedures that I've done before. This book doesn't try to be impressive or pompous. It is written simply — listing clear concise instructions on how to do things, in step-by-step form.

Let me explain how it's organized. It can't be read through from beginning to end. It is meant as a reference book that can be kept next to your computer system.

The first thing you'll see after this preface is not a table of contents, but an index. The index is the road map that points out the locations of the answers. This index is not a typical index. Typical indexes (indices?) are compiled by orangutans after the book is edited. They're only marginally usable as *all* references to the subject are listed, in page number order. Our index lists the most important reference first, followed by other references in decreasing importance. Also, the subject is listed in many forms. If you want to know how to check out continuity of a printer cable, for example, you'll find it under "Buzzing out a cable BOCH," "Cable, buzzing out BOCH," "Cable, continuity test BOCH," and other references. The reason? To reduce your search time.

The 4-letter code after each reference (such as "BOCH") is the "key" to the procedure. It helps in your visual search and is close to the initials of the procedure heading. All procedures are listed in alphabetical order, by this 4-letter code.

To pack as much information as possible into these pages, we've used the "pyramid" approach. "CompuServe, Using," for example, references a subordinate entry called "Modem, How To Use," which references further subordinate entries such as "Modems, What Are They?" and "RS-232-C, What Is It?" Each entry clearly indicates other entries that are referenced by key word, so you'll have no trouble following the entries down to the last instruction. You won't see the pyramid in the structure of the book, but you will notice the subordinate references as you follow the instructions.

About humor... I hope you don't mind. I've included some humor in the book when appropriate. If this offends you, you are not emotionally equipped to use a microcomputer system, and you'll probably go mad, anyway. After all, what's funnier than trying to accomplish something useful on a microcomputer (or any computer) and not being able to find that terminating character?

Thanks to my wife Janet for her help in proofing and technical corrections.

To Murphy, the maintenance man at One Tandy Center. Thanks for the inside information.

William Barden jr. — 1983

Index

notes

List of Procedures by Keyword

CCPA	Color Computer Pixel Addressing
CD13	Cassette DEBUG, Models I and III
CDFH	CREATEing a Disk File, Model I/II/III
CDFL	Copying Disk Files, Model I/III LDOS
CDHT	Clock Display, How to Use, TRSDOS/LDOS, Models I/II/III
CDOC	Changing the Display Offset
CFAD	Copying a File to Another Diskette, Model I/II/III/Color Computer TRSDOS
CFBD	Converting from Binary to Decimal
CFCS	Converting the First Character of a String to Numeric in BASIC, Most Systems
CFDB	Converting from Decimal to Binary
CFDH	Converting from Decimal to Hexadecimal
CFFT	/CMD File Format, Model I/II/III
CFHD	Converting from Hexadecimal to Decimal
CFOW	Cursor, Finding Out Where It Is In BASIC
CFSD	Copying a File to Same Diskette, Model I/II/III/Color Computer TRSDOS
CHON	Commands, Help On, Model II/III TRSDOS
CHTP	Columns, How to Put Things In, in BASIC
CLCN	Cleaning Connectors
CLTD	Clearing the Display in BASIC, All Systems
CMD1	Calling a Machine Language Program in Disk BASIC, Model I or Model III TRSDOS/LDOS
CMEC	Calling a Machine Language Program in Extended Color BASIC, Color Computer
CMHT	Clearing Memory, Model II/III TRSDOS, Model I/III LDOS
CML1	Calling a Machine Language Program in Non-Disk BASIC, Model I or Model III
CMLC	Calling a Machine Language Program in Color BASIC, Color Computer
CNER	CN Error
CPCA	Centronics (Printer) Cables, Model I/II/III
CPM3	Compressing Programs, Model III TRSDOS
CPSU	CompuServe, Using
CPWR	Conditioning Power
CRPI	Cassette Recorder Plugs, Insertion of
CSFT	Clearing the Screen from TRSDOS, Model II/III
CSHT	Comparing Strings in BASIC
CSNV	Converting BASIC Strings to Numeric and Vice Versa
CSUM	Clock Speed Up Modifications
CTLC	Cassette Tape Loading Difficulties, Model I/III/Color Computer
CUSE	CHR$ Use, BASIC, Most Systems
CWTC	Commands, What TRSDOS/LDOS Commands Are There?, Model I/II/III
D1TW	DEVICE, Model II TRSDOS, What is It?
DADF	Deleting Many Disk Files, How to, Model II/III TRSDOS
DCJS	Date Conversion in BASIC, Julian/Standard Format, Model III TRSDOS
DDER	DD Error
DDWA	Data Diskettes, What Are They?
DEHM	Disk Errors, How Many are Too Many?
DFAH	Disk File Access, How to Speed Up
DFBH	DIR from Inside Disk BASIC, Model III TRSDOS
DFDA	Disk File, Deleting, All Systems
DFOT	Date, Finding Out Today's, Model I/II/III Disk Systems
DI13	Discrete Inputs for the Model I or III
DIBP	Date, in BASIC Program, Model II
DICC	Discrete Inputs for the Color Computer
DIDI	Directory, TRSDOS/LDOS, All Systems
DINS	Diskettes, Inserting
DLDB	DEBUG, Loading from Disk BASIC, Model I TRSDOS, Model I/III LDOS
DLIB	Deleting Lines in BASIC, All Systems
DLM1	Directory Listing, Model I, TRSDOS
DLM2	Directory Listing, Model II, TRSDOS

DLM3	Directory Listing, Model III, TRSDOS
DLSC	Digital Logic, A Short Course
DLWS	Disk Light, When Should It Come On?
DMT2	Dumping Memory to a Disk File, How to, Model II TRSDOS
DMT3	Dumping Memory to a Disk File, How to, Model III TRSDOS
DMTL	Dumping Memory to a Disk File, How to, Model I/III LDOS
DMTT	Dumping Memory to a Disk File, How to, Model I TRSDOS
DPHU	Double-Precision Variables in BASIC, How to Use
DSBC	Drawing Shapes in BASIC, Color Computer
DSDS	Date, Setting, Model I/II/III, TRSDOS/LDOS
DSHL	Disk Space, How Much Left?, TRSDOS, All Systems
DSPR	Dumping the Screen to the Printer in BASIC, Model I/III
DSRC	DATA Statements and Related Commands, BASIC, All Systems
DT1U	DEBUG, TRSDOS/LDOS, Model I/III, Using
DUFS	Diskettes, Using Flip Side
DWDK	DEBUG, Why Do I Keep Entering, Model I/II TRSDOS, Model I/III LDOS
DWPR	Diskette, Write Protecting
DZDO	Drive 0, Drive 1, Which Is It?
DZER	Divide by 0 Error
EAIM	EDTASM+, Color Computer, Assembling Into Memory
ECCE	Expressions, Color Computer EDTASM+
ECIS	Embedded Codes in SCRIPSIT
EDAN	EDAS, Notes
EDCE	EDTASM+, Color Computer, Using Editor and Assembler
EMBH	Edit Mode, BASIC, Most Systems
EPHT	Ending a BASIC Program, How to, All Systems
EPWT	Electronic Parts, Where to Get
ETIB	Error Trapping in BASIC, Some Systems
EYBP	Erasing Your BASIC Program
FBDF	Functions, BASIC, Defining, Some Systems
FBWT	500-Baud and 1500-Baud, For Model III Cassette, BASIC When to Use
FCER	FC Error
FDER	FD Error
FHBF	500-Baud or 1500-Baud for Model III Cassette, BASIC, Selecting
FHOF	Failures, How Often?
FHSF	Files, How System Finds
FNFW	File Not Found When the Directory Says It's There!
FNMH	File Names, How to Use
FSG2	Video/Graphics Worksheet, Model II
FSGC	Video/Graphics Worksheet, Color Computer
FSGW	Video/Graphics Worksheet, Model I/III
FTLO	FOR . . . TO Loops, BASIC, Ok to Break Out? All Systems
FTST	FOR . . . TO . . . STEP, BASIC, All Systems
GAPC	GET and PUT Use in BASIC Graphics, Color Computer
GC13	Graphics Characters, Model I/III
GHS1	Graphics, High-Speed, Model I/III
GMIC	Graphics Modes in the Color Computer
GMPC	Graphics Mapping in the Color Computer
GPAR	Graphics Architecture, Color Computer
HASA	Amount of Storage in a BASIC Array, How to Find
HMF1	How Many Files, Model I/III
HNIB	Hexadecimal Notation in BASIC Some Systems
HTDS	Displaying at any Screen Location in BASIC, All Systems
HTGB	Back to Level II BASIC from Disk BASIC, Model I TRSDOS
ICIG	Integrated Circuit, Inserting
ICRS	Integrated Circuit, Removing a Soldered

SCOF	/CIM on Files, What Does It Mean?
SDFB	Sequential Disk Files, Using
SDWA	System Diskettes, What Are They?
SDTD	SETting a Device to a Driver, Model I/III LDOS
SFLO	Strings, Finding Length of, BASIC, Most Systems
SHTC	Scrolling, How to Control, Model III
SHTO	Soldering, How to
SHTU	Strings, BASIC, How to Use
SIPS	Semicolon, in BASIC PRINTs
SNER	SN Error
SPHT	Spooling, How to Use, Model I/III LDOS
SPHU	Single-Precision Variables in BASIC, How to Use
SRBH	SET/RESET in BASIC, How to Use, Model I/III
SSA3	Sorting a String Array in BASIC, Model III TRSDOS, Model I/III LDOS
SSFA	String Space, Finding Amount Left
STAC	Stack, Description
STER	ST Error
STRD	STRING$ Command BASIC, Most Systems
STWD	/TXT on Files, What Does It Mean?
SUBB	Subroutines, BASIC, All Systems
SUBP	Speeding Up Your BASIC Programs
SUCG	Setting up for Color Computer Graphics in BASIC
SVDG	Setting the VDG Modes in the Color Computer
SWHU	Slit and Wrap, How to Use
TBCC	Tokens in BASIC, Color Computer
TCMH	32-Character Mode, How to Switch to in BASIC, Model I/III
TCML	Transferring Control to a Machine Language Program from BASIC, Model I/III Non-Disk
TCNU	Two's Complement Numbers, Using
TGBB	TRSDOS, Getting Back to From BASIC, Model II
THT3	TRSDOS, How to Load, All Systems
TIBP	Time, in BASIC Program, Most Systems
TM13	Tokens in BASIC, Model I/III
TMER	TM Error
TMTW	Tokens in BASIC, Model II
TOCH	Turning on the Computer, How to
TRHT	Tracing, BASIC Programs
TRUS	TRACE, Using, Model I TRSDOS, Model, I/III LDOS
TSAL	Table Size in Assembly Language
TSPD	To Stop the Display in BASIC, All Systems
TSRT	TIME, Setting the Real-Time-Clock TRSDOS/LDOS, Models I/II/III
TSTP	To Stop the Program
UEER	UE Error
ULER	UL Error
UTFB	Using Trigonometric Functions in BASIC
VAIF	Visible and Invisible Files, What Are They?
VMHT	Voltmeter, How to Use
VTTO	VERIFY, to Turn On and Off, Model I TRSDOS, Model I/III LDOS, Color Computer DOS
VTTT	VERIFY, to Turn On and Off, Model II TRSDOS
VWDI	VERIFY – What Does it Do?
WCCD	"Wild-Card" Copies of Disk Files, Model III TRSDOS
WWHT	Wire-Wrapping, How to
Z8AF	Z-80 Address Format
ZUEC	ZBUG, Using, EDTASM+, Color Computer

ACHT

Acoustic Coupler, How to Use

1. Read procedure MWAT if you know nothing about modems.

2. Are you sure you have an acoustic coupler (with two cups for the telephone headset)? If so, continue. If not, go to procedure MHTU.

3. Connect a standard 25-conductor ribbon cable between the 25-pin RS-232-C connector on the modem and the 25-pin RS-232-C connector of the Model I, II (B), or Model III. This cable is available from Radio Shack (26-1408) and other electronics or computer stores. Connect a special cable (RS 26-3014) between the Color Computer RS-232-C port (4-pin DIN, see BSCC) and the 25-pin RS-232-C connector of the modem; some Radio Shack modems will have special 4-pin DIN plugs, and you can use RS cable 26-3020.

4. Model I: setup your COMM/TERM and Sense Switches as described in procedure RHS1. Model II: setup the RS-232-C interface as described in the RS manual. Model III: setup the RS-232-C interface as described in procedure RHS3. Color Computer: setup the RS-232-C interface as described in the RS manual.

5. You're now ready to dial a Bulletin Board system, CompuServe, or other data communications system. Setup your modem this way:

● If you are originating the call, set the modem switch for Originate/Answer to "Originate." Sometimes this switch will be labeled "O/A."

● Set the modem switch for "Full/Half" to "Full" if your program uses full duplex (see RSWI) or "Half" if your program uses half duplex. Typical use is full duplex to "echo-back" characters transmitted.

6. Load and start your data communications program. Simple programs will now be waiting with a blank screen, ready to receive data and display it, and to transmit data from the keyboard.

7. Dial up the number of the Bulletin Board or network while listening on the phone. After the phone is answered, you hear a pause followed by a high-pitched whine of the "carrier" frequency. Take your time (you have a minute or so), and carefully place the telephone handset in the cups of the acoustic coupler, making certain that they are securely encased in the rubber cups to blanket room noise. Note the proper position of the handset. If you turn it around, the coupler will simply not work.

8. You should now see the prompt message of the Bulletin Board or network on the screen. If you do not, try typing ENTER a few times. If you still see nothing, check the "carrier" light on the modem. If it is off, the other system has "disconnected" you. (It will "time out" if it does not get the proper response in time). Review the steps above, and try again with another system (preferably). If you still have trouble, refer to procedure RSH1 or RSH3. If you do see data on the screen, but it's garbled, go to procedure RSH1 or RSH3 to check the word length, stop bit, and baud rate settings. If you see meaningful data, continue with the procedure for Bulletin Boards (BBUS) or CompuServe (CPSU).

ACPR

AC Plug, Rewiring

Want to cut off that long cord from your TRS-80 to get a neater computer room appearance? You can do it easily, and it won't void your warranty.

1. You'll need a 3-connector AC male plug, available from any hardware store. Do not use a 2-connector plug!

2. Refer to Figure ACPR-1. Most AC power cords have three wires inside, colored white, green, and black. The green wire is always ground. The white and black wires are "polarized" and go to definite sides of the plug that goes into the wall as shown.

3. Cut the cord at a convenient length using diagonal pliers or even scissors.

4. Strip the outer covering on the cord (the rubber part) so that about 1 inch of the 3 wires are exposed. You'll find some fiber material present, and you can cut this off, leaving only the 3 wires. Make certain that you don't cut the outer covering too deeply – just enough to allow you to pull off the covering without exposing bare wire on the three internal wires.

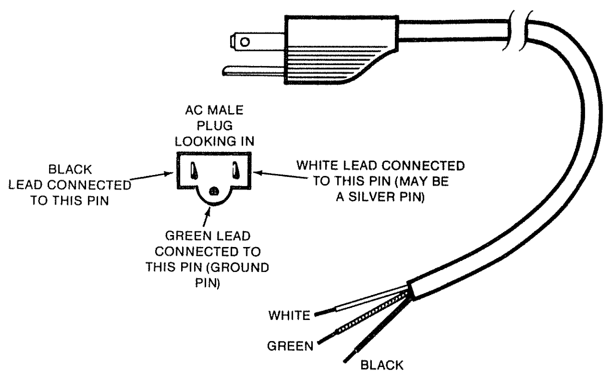
5. Strip about 1/2 inch off each of the three wires. Roll the strands of each wire together so that they form a compact wire without loose ends. Form each of the three wires into a "U" shape. If you're a perfectionist, you can solder each wire into a permanent "U" shape (see SHTO) although this isn't really necessary.

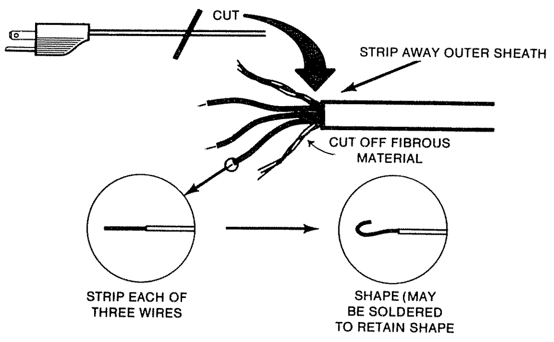
6. Disassemble the 3-connector plug. Here's a description of a typical plug: there are 3 screws on the face of the plug. Unscrew these and pull the two parts of the plug apart. The "hood" of the plug has two screws that clamp around the "line cord." Loosen these screws and slide the hood up the cut line cord that attaches to the computer equipment.

7. Connect the 3 wires to the three-connector male AC plug as shown. The plug may have a white connector, indicating the proper connection for the white wire. The green wire should always go to the round ground connector. You may have to unscrew the screw connections completely off the plug to connect each wire. The "U" shape should follow the direction of screw rotation when the screw is screwed *in*, to avoid "splaying" the wires.

8. Reconnect the "hood" and body of the plug, and tighten the hood screws to clamp around the line cord. This will provide "strain relief" so that there is no stress on the 3 wires that would act to pull them loose.

Figure ACPR-1 – *AC Plug Wiring*


AC MALE PLUG LOOKING IN
BLACK LEAD CONNECTED TO THIS PIN
WHITE LEAD CONNECTED TO THIS PIN (MAY BE A SILVER PIN)
GREEN LEAD CONNECTED TO THIS PIN (GROUND PIN)
WHITE
GREEN
BLACK


CUT
STRIP AWAY OUTER SHEATH
CUT OFF FIBROUS MATERIAL
STRIP EACH OF THREE WIRES
SHAPE (MAY BE SOLDERED TO RETAIN SHAPE

ADC2
Attributes of a Diskette, Changing TRSDOS, Model II

See ADFC if you want to change the attributes of a disk *file.* This section is for changing the diskette master password, and a "blanket" LOCK, UNLOCK of disk files. (See PWDS for a description of passwords).

To change the master password:

```
PROT :n OLD=oldpass, NEW=newpass
```

The :n parameter is a drive specification denoting the disk drive (:0 – :3). The colon is optional. The OLD parameter is the current master password of the diskette (the standard password on most diskettes is PASSWORD). The new parameter is the new master password to be used on the diskette. From 1 to 8 alphanumeric characters can be used for the new password.

To remove all access and update passwords from user visible, non-system files, use:

```
PROT :n OLD=oldpass, UNLOCK
```

If the OLD password is correct, TRSDOS will "unlock" all visible, non-system files, a powerful command. To force all access and update passwords to the current disk master password, use:

```
PROT :n OLD=oldpass, LOCK
```

If the OLD password is correct, TRSDOS will use the master password for all user file access and update passwords. The NEW option and LOCK or UNLOCK can be used at the same time. If LOCK or UNLOCK is omitted, the file protection on the diskette is left unchanged.

ADCL
Attributes of a Diskette, Changing LDOS, Model I/III

See ADFL if you want to change the attributes of a disk *file.* This section is for changing the diskette name, master password, and a "blanket" LOCK, UNLOCK of disk files. (See PWDS for a description of passwords).

To change the diskette name, use the following format:

```
ATTRIB :n (NAME = "newname",MPW = "password")
```

The :n is the drive specification (:0 – :3) and is optional; if none is used, :0 is assumed. The NAME parameter is the new name of the disk and the MPW, or Master Password, is the current master password. The MPW parameter is not required if PASSWORD is the current master password. To change the master password:

```
ATTRIB :n (PW="newpass",MPW="password")
```

The name and password can be changed in the same command. To remove all access and update passwords from user visible, non-system files, use:

```
ATTRIB :n (UNLOCK,MPW="password")
```

The MPW parameter is not necessary if the master password is PASSWORD. This command "unlocks" all visible, non-system files, a powerful command.

To force all access and update passwords to the current disk master password, use:

```
ATTRIB :n (LOCK)
```

ADCT
Attributes of a Diskette, Changing TRSDOS Model I/III

See ADFC if you want to change the attributes of a disk *file*. This section is for changing the diskette master password, and a "blanket" LOCK, UNLOCK of disk files. (See PWDS for a description of passwords.) To change the master password:

```
PROT :n (PW)
```

The :n parameter is a drive specification denoting the disk drive (:0 – :3). TRSDOS will prompt you for the old password (PASSWORD initially on most diskettes) and then for the new password that will replace it.

Model I users only: To remove all access and update passwords from user visible, non-system files, use:

```
PROT :n (UNLOCK)
```

TRSDOS will prompt you for the master disk password, and, if you've entered it correctly, will "unlock" all visible, non-system files, a powerful command. To force all access and update passwords to the current disk master password, use:

```
PROT :n (LOCK)
```

TRSDOS will prompt you for the master disk password, and, if you've entered it correctly, will use the password for all user file access and update passwords. The PW option and LOCK or UNLOCK can be used at the same time.

ADFC
Attributes of a Disk File, Changing TRSDOS Model I/II/III

The ATTRIB command lets you change the visible/invisible status of a file (see VAIF), change the access/update passwords (see PWDS), and change the access level. To change a visible file to invisible, do

```
ATTRIB name (I) (Model I/III)
ATTRIB name I  (Model II)
```

where "name" is a file name (see FNMH). To change back to visible status, use N in place of I, except for Model I. For TRSDOS Model I do a COPY to a new file (deucedly inconvenient).

To assign a new access or update password, do

```
ATTRIB name (ACC=password)    (Model I/III)
ATTRIB name ACC=password      (Model II)
```

or

```
ATTRIB name (UPD=password)    (Model I/III)
ATTRIB name UPD=password      (Model II)
```

or use both options at once. The "password" may be 1 to 8 characters.

To remove access and/or update passwords, use the (ACC= ,) or (UPD= ,) form of ATTRIB. The access or update password will be set to blanks (no password). To change the access level do

```
ATTRIB name (PROT=level)       (Model I/III)
ATTRIB name PROT=level         (Model II)
```

where "level" is KILL, RENAME, WRITE, READ, or EXEC. (Or NONE for Model II). The access level is defined as follows:

FULL (III)	Full access, no protection
KILL	Total access
RENAME (I)	Rename, write, read and execute
NAME (III)	Rename, write, read, and execute
WRITE	Write, read, and execute
READ	Read the file and execute, may be listed
EXEC	Execute the file only, impossible to obtain a listing
NONE (II)	ultimate paranoia – no access

You can use one or more parameters in each ATTRIB command.

ADFL
Attributes of a Disk File, Changing LDOS Model I/III

The ATTRIB command lets you change the visible/invisible status of a file (see VAIF), change the access/update passwords (see PWDS), and change the access level. To change a visible file to invisible or vice versa, do

```
ATTRIB name (VIS)
```

or

```
ATTRIB name (INV)
```

where "name" is a file name (see FNMH). To assign a new access or update password, do

```
ATTRIB name (ACC=password)
```

or

```
ATTRIB name (UPD=password)
```

or use both options at once. The "password" may be 1 to 8 characters.

To remove access and/or update passwords, use a "null" password (no characters after the equals sign).

```
ATTRIB name (UPD=,ACC=)
```

To change the access level do

```
ATTRIB name (PROT=level)
```

where "level" is ALL, FULL, KILL, NAME, WRITE, READ, or EXEC. The access level is defined as follows:

ALL	Total access
FULL	Total access
KILL	Total access except resetting attributes
NAME	Rename, write, read, and execute
WRITE	Write, read, and execute
READ	Read the file and execute, may be listed
EXEC	Execute the file only, impossible to obtain a listing

Notes: You can use any combination of the above parameters; there are no defaults and nothing will change in the file attributes if you do not specify a parameter. Abbreviations are U, A, P, V, I and AL, FU, KI, RE, WR, NA, and EX.

ADFW
ASCII Characters, What Are They?

ASCII refers to the code used to represent alphabetic, numeric, or special characters. It is a "7-bit" code, with the upper, or most significant bit not used. This means that values of 0 through 127 are valid ASCII characters. ASCII code is shown in Table ADFW-1.

The ASCII codes lower than 32 decimal (**20H**) are called "control codes" because they are set aside for special functions such as "line feeds" (ejecting paper from a printer), "carriage returns" (start at beginning of line) or "move cursor."

All the TRS-80 systems use ASCII in BASIC and other applications to represent alphabetic, numeric, and special characters.

Table ADFW-1 – *ASCII Codes used on the TRS-80s*

CC* Code	I/II/III Code	Character
	Ø	
	1	BREAK
	2	
	3	
	4	
	5	
	6	
	7	
	8	LEFT ARROW
	9	RIGHT ARROW
	1Ø	DOWN ARROW
	11	
	12	
	13	ENTER
	14	
	15	
	16	
	17	
	18	
	19	
	2Ø	
	21	
	22	
	23	
	24	
	25	
	26	
	27	
	28	
	29	
	3Ø	
	31	CLEAR
32/96	32	blank
33/97	33	!
34/98	34	"
35/99	35	#

CC* Code	I/II/III Code	Character
36/1ØØ	36	$
37/1Ø1	37	%
38/1Ø2	38	&
39/1Ø3	39	'
4Ø/1Ø4	4Ø	(
41/1Ø5	41	)
42/1Ø6	42	*
43/1Ø7	43	+
44/1Ø8	44	,
45/1Ø9	45	-
46/11Ø	46	.
47/111	47	/
48/112	48	Ø
49/113	49	1
5Ø/114	5Ø	2
51/115	51	3
52/116	52	4
53/117	53	5
54/118	54	6
55/119	55	7
56/12Ø	56	8
57/121	57	9
58/122	58	:
59/123	59	;
6Ø/124	6Ø	<
61/125	61	=
62/126	62	>
63/127	63	?
Ø/64	64	@
1/65	65	A
2/66	66	B
3/67	67	C
4/68	68	D
5/69	69	E
6/7Ø	7Ø	F
7/71	71	G
8/72	72	H
9/73	73	I
1Ø/74	74	J
11/75	75	K
12/76	76	L
13/77	77	M
14/78	78	N
15/79	79	O
16/8Ø	8Ø	P
17/81	81	Q
18/82	82	R

CC* Code	I/II/III Code	Character
19/83	83	S
2Ø/84	84	T
21/85	85	U
22/86	86	V
23/87	87	W
24/88	88	X
25/89	89	Y
26/9Ø	9Ø	Z
27/91	91	↑
28/92	92	↓
29/93	93	←
3Ø/94	94	→
31/95	95	_
	96	@
	97	a
	98	b
	99	c
	1ØØ	d
	1Ø1	e
	1Ø2	f
	1Ø3	g
	1Ø4	h
	1Ø5	i
	1Ø6	j
	1Ø7	k
	1Ø8	l
	1Ø9	m
	11Ø	n
	111	o
	112	p
	113	q
	114	r
	115	s
	116	t
	117	u
	118	v
	119	w
	12Ø	x
	121	y
	122	z
	123	{
	124	|
	125	}
	126	~
	127	

*Color Computer code shown is "non-inverted" followed by "inverted" code. Inverted is black on green and non-inverted is green on black.

ADIC
Analog-to-Digital Inputs, Color Computer

The joystick inputs of the Color Computer may be used for four channels of an analog-to-digital converter. The inputs would represent a voltage analog of a real-world quantity, such as temperature. A simple temperature sensor is shown in Figure ADIC-1.

The temperature sensor is a thermistor, a device whose resistance changes with temperature. The thermistor and 15K ohm resistor form a "voltage divider." The voltage at the junction of the two is read into the right joystick X channel by the JOYSTK(0) command. The statement

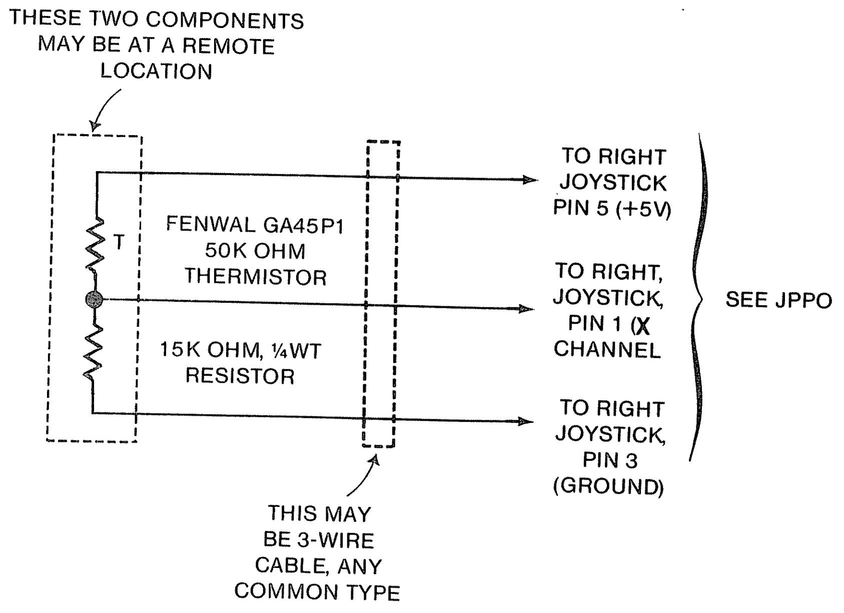
```
100 A=JOYSTK(0)
```

for example, sets variable A to a value of 0-63 depending upon the voltage of the junction, which will be close to 0 volts (A = 0) through close to +5 volts (A = 63), depending upon the resistance of the thermistor, which is dependent upon temperature.

You can use this same scheme to read in the other three channels by using JOYSTK(1), JOYSTK(2) and JOYSTK(3), with the devices attached to the proper pins. Remember that JOYSTK(0) must first be executed before reading any of the other channels.

The voltage input to the X or Y channel may be derived from other types of circuits, such as "op-amps," and may represent a variety of other "transducer" inputs, such as pressure, light intensity and position. For more information on reading real-world inputs, see my book *TRS-80 Model I, III, and Color Computer Interfacing Projects,* Howard W. Sams, Publisher.

Figure ADIC-1 – *Simple Temperature Sensor*


THESE TWO COMPONENTS MAY BE AT A REMOTE LOCATION
T
FENWAL GA45P1 50K OHM THERMISTOR
15K OHM, ¼WT RESISTOR
TO RIGHT JOYSTICK PIN 5 (+5V)
TO RIGHT, JOYSTICK, PIN 1 (X CHANNEL
TO RIGHT JOYSTICK, PIN 3 (GROUND)
SEE JPPO
THIS MAY BE 3-WIRE CABLE, ANY COMMON TYPE

AECE
Assembler Error Diagnostics, Color Computer EDTASM+

BAD LABEL:	Invalid label characters. Redo label.
BAD MEMORY:	You did an in-memory assembly that attempted to overwrite system memory, the edit buffer or symbol table, the protected area set by USRORG, or was over the top of RAM. See EAIM.
BAD OPCODE:	Use only valid 6809E opcode or pseudo-opcode mnemonics.
BAD OPERAND:	Illegal operand, for example, `LDX ,,X`
BYTE OVERFLOW:	Operand too large for field, as in `TABLE FCB ADDRESS generate address`
DP ERROR:	The high order byte of an operand does not match the value set by the last SETDP command (see POCE).
EXPRESSION ERROR:	Invalid expression in operand.
MISSING END:	No END statement. You'd think the assembler would be smart enough . . .
MISING INFOR-MATION:	(There's a little joke here . . .)

Missing delimiter in an FCC, or no label on a SET or EQU.

MISSING OPERAND: One or more operands missing.

MULTIPLY DEFINED SYMBOL: Label of line was used somewhere else. First definition is used.

REGISTER ERROR: No register in a PSH/PUL or a register specified more than once in a PSH/PUL, or improper registers in an EXG or TFR instruction.

UNDEFINED SYMBOL: Operand symbol has not been defined. Define by new label.

Figure AECE-1 illustrates a catastrophic assembly containing these types of errors.

Figure AECE-1 – *Assembler Error Diagnostics, 6809E*

```
BAD LABEL
ØØ1ØØ DIVID%  PSHS    X,A      DIVIDEND, DIVISOR
BAD OPCODE
ØØ11Ø         CLRX             CLEAR 1/2 OF DIVIDEND
BAD OPERAND
ØØ12Ø         LDB     +1,SS    GET MSB OF DIVIDEND
UNDEFINED SYMBOL
ØØØØ 8D   FE      ØØ13Ø         BSR               DO 8 DIVIDES
ØØØ2 E7   61      ØØ14Ø         STB      +1,S     REPLACE FIRST HALF
ØØØ4 E6   62      ØØ15Ø DIV9ØØ  LDB      +2,S     GET LSB OF DIVIDEND
MULTIPLY DEFINED SYMBOL
UNDEFINED SYMBOL
ØØØ6 8D   F8      ØØ16Ø DIV9ØØ  BSR      DIVIDE   DO 8 DIVIDES
ØØØ8 E7   62      ØØ17Ø         STB      +2,S     REPLACE 2ND 1/2
REGISTER ERROR
BAD OPERAND
ØØ18Ø         PULS              DISCARD DIVISOR
ØØØC 39           ØØ19Ø          RTS              RETURN
MISSING END
ØØØØ9 TOTAL ERRORS

DIV9ØØ  ØØØ4    M
DIVIDE  ØØØØ    U
DO      ØØØØ    U
```

AEDI
Assembler Error Diagnostics, Model I/II/III Assemblers

Fatal Errors: No object code generated for line:

BAD LABEL: Invalid label characters. Redo label.

EXPRESSION ERROR: Invalid expression in operand.

ILLEGAL ADDRESSING MODE: Illegal operand, for example,

```
JR P,START          ;jump on positive
```

ILLEGAL OPCODE: Use only valid Z-80 opcode mnemonics.

MISSING INFORMATION: Operand missing as in

```
JP                  ;go to start
```

Warnings: Object code generated but may not be correct:

BRANCH OUT OF RANGE: JR type branch was more than 129 bytes forward or 126 bytes back. Make a JP or move branch point closer.

FIELD OVERFLOW: Operand too large for field, as in

```
TABLE  DEFB   ADDRESS  ;generate address
```

MULTIPLY DEFINED SYMBOL: Label of line was used somewhere else. First definition is used.

MULTIPLE DEFINITION: Reference of a multiply-defined symbol.

NO END STATEMENT: Put one in.

UNDEFINED SYMBOL: Operand symbol has not been defined. Define by new label.

Figure AEDI-1 illustrates a catastrophic assembly containing these types of errors.

Figure AEDI-1 – *Assembler Error Diagnostics, Z-80*

```
8ØØØ            ØØ1ØØ           ORG     8ØØØH
Bad label
ØØ11Ø #TRT      LD      A,ØFFFØH
8ØØØ B7         ØØ12Ø           OR      A
Illegal opcode
ØØ13Ø           CMP     A,B
Illegal addressing mode
ØØ14Ø           JR      P,START
Branch out of range
8ØØ1 18FE       ØØ15Ø           JR      9ØØØH
Multiple definition
Undefined symbol
8ØØ3 3EØØ       ØØ16Ø RESET     LD      A,TABLE
Field overflow
8ØØ5.Ø6ØC       ØØ17Ø           LD      B,TABLE1+3
Multiple definition
Undefined symbol
Branch out of range
8ØØ7 18FE       ØØ18Ø RESET     JR      FINISH
Field overflow
8ØØ9 E8         ØØ19Ø TABLE1    DEFB    1ØØØ
8ØØA 7Ø11       ØØ2ØØ           DEFW    7ØØØØ
No end statement
```

AEDP
Automatic Execution of a Disk Program on Power Up

Use the AUTO command. This command lets you specify a DOS command that will be executed on power up or reset. The command will be recorded permanently on diskette. The action is just as if you had loaded the system and then typed in the corresponding command.

The usual use of AUTO is to specify a program that will be executed after "booting." If you had a SCRIPSIT diskette, for example, and wanted to automatically go into SCRIPSIT after booting, you could say

```
AUTO SCRIPSIT
```

However, you could also specify any other TRSDOS or LDOS command, such as

```
AUTO DIR
```

Model I TRSDOS users skip this section: The most powerful feature of AUTO allows you to execute a DO (JCL) file. This will start a sequence of commands that take the place of command lines input from the keyboard. The format of AUTO for this operation is

```
AUTO DO name
```

where "name" is a DO file name.

To reset the AUTO capability: Perform an

```
AUTO
```

alone. This will reset the AUTO command on disk.

To reboot a disk and disable the AUTO function: Hold down the ENTER key as you reboot. This will bring you into the DOS prompt, but will not remove the AUTO command from the disk.

LDOS users only: A special form of AUTO disables the ENTER disable and also disables the BREAK key. The form is:

```
AUTO *command
```

You can re-enable the BREAK key by SYSTEM (BREAK =ON) after the AUTO sequence.

AEU1
Assembler Expressions, Using, Models I/II/III Assemblers

Expressions are used to define operands in assembler source lines. Limited arithmetic processing can be used.

The plus (+) and minus (–) signs are used in BASIC, for adding two terms or for the sign of the value:

```
TABLE   DEFW    START+2ØØH    ;start + 512 bytes
        DEFW    END-56        ;end - 56 bytes
        DEFW    +111          ;111 value in 16 bits
        DEFW    -45           ;-45 value in 8 bits
```

The result of the expression must be small enough to fit in the data type. The result for DEFB must be 0 to 255; a quirk of the RS EDTASM assemblers is that negative values are not allowed for DEFB. Use the hex equivalent instead:

```
TABLE   DEFB    ØFFH          ;-1
```

Logical ANDs are defined by the ampersand (&). They work the same way as a logical AND in BASIC:

```
TABLE   DEFW    START&ØFFFØH ;drop 8 ls bits
```

A shift operator is defined by a less than sign (<). If the value following the less than is negative, a logical right shift of n bits is done (zeroes fill on left). If the value following the less than is positive, a logical right shift is done (zeroes fill on left):

```
TABLE   DEFW    START<8       ;if ABCDH, now CDØØH
        DEFW    START<-8      ;if ABCDH, now ØØABH
```

Again, the result must be small enough to be held in the data type involved.

Figure AEU1-1 shows examples of operations.

See EDAS notes for extended operations on the EDAS assembler (EDAN).

Figure AEU1-1 – *Assembler Expressions*

```
ØØ1ØØ ; ASSEMBLER EXPRESSIONS
8123            ØØ11Ø           ORG     8123H
ØØØD            ØØ12Ø CR        EQU     13
8123 C1         ØØ13Ø TABLE     DEFB    128+'A'         ;ADDITION
8124 57         ØØ14Ø           DEFB    1ØØ-CR          ;SUBTRACTION
8125 E8Ø3       ØØ15Ø           DEFW    +1ØØØ           ;POSITIVE SIGN
8127 18FC       ØØ16Ø           DEFW    -1ØØØ           ;NEGATIVE SIGN
8129 FF         ØØ17Ø           DEFB    -1              ;ERROR HERE
812A 2Ø81       ØØ18Ø           DEFW    TABLE&ØFFFØH    ;LOGICAL AND
812C ØØ23       ØØ19Ø           DEFW    TABLE<8         ;SHIFT LEFT
812E 81ØØ       ØØ2ØØ           DEFW    TABLE<-8        ;SHIFT RIGHT
No end statement
ØØØØØ Total errors
```

AFWA
ASCII Files, What Are They?

Read ADFW if you don't know what ASCII characters are.

ASCII files are made up of ASCII characters from 32 decimal (**20H**) through 127 decimal (**7FH**), in addition to special control codes such as carriage return (**0DH**). No other codes are used. This means that the file is "displayable" on the video screen or "printable" on the system line printer. ASCII files take up more space than an "encoded" type of file, but can easily be examined by display or printing.

BASIC, SCRIPSIT, and other programs work with a specially encoded type of file that saves space. One BASIC "token" byte (see TM13, TMTW, TBCC for I/III, II, and Color Computer respectively), for example, may repiace six bytes or more of a command name.) These programs, however, sometimes also offer the option of writing ASCII files. The ASCII files use only ASCII characters, and no special codes.

ASCII files are a "standard" format, and for this reason, certain TRSDOS or LDOS commands, such as APPEND, will only work with ASCII files.

AGTU
Arrays in BASIC, Using

Arrays are collections of data called by the same name. Suppose that you wanted to record 100 names. You could call the first A$, the second B$, and so forth, up to DA$, or some other unique name. It would be much easier, though, to establish an array called NM$ and allocate 100 spaces for the names. This is done by

```
100 DIM NM$(99)
```

This command sets aside 100 entries in a string array called NM$, for name. The first entry is referenced by NM$(0), the second by NM$(1), and so forth, up to NM$(99). Note that the last entry "index" is one less then the size of the array – 100 entries, but the last is NM$(99).

You could also use a *numeric array*, such as

```
100 DIM AG(99)
```

which would set up a 100-entry numeric array called AG, holding 100 age values and referenced by AG(0) through AG(99).

The above arrays were "one-dimensional" arrays, or simply lists of data.

'Two-dimensional" arrays are like a checkerboard – the entries are referenced by x and y coordinates. A two-dimensional array holding points of a graph might be

```
100 DIM GR(49,49)
```

In this case there are 50 times 50 entries, or 2500 entries in the array. The first is referenced by GR(0,0), and the last by GR(49,49).

Can you have more than two dimensions? Yes, 3 or more, if the need arises, and the need is not uncommon in mathematical processing.

What about the data in arrays? The DIM(ension) statement just allocates the space for the array – you have to initialize it with data (see INAR). If the data you put in is not in order, that's the way it'll stay. Here's an example of the way an array would be filled with 100 names:

```
100 DIM NM$(99)
110 FOR I=0 TO 99
120 INPUT NM$(I)
130 IF NM$(I)="ZEGLOVITZ" THEN GOTO 160
140 I=I+1
150 NEXT I
160 PRINT "DONE OR ZEGLOVITZ AGAIN"
```

AIOF
Assembling, Is Object File In Memory After Assembling?

Usually not. You must assemble the object to a disk or cassette file and then load it from TRSDOS/LDOS or from cassette by SYSTEM or CLOADM.

Exception: Using an in-memory assembler such as EDAS or the Color Computer EDTASM+ with an in-memory assembly specified. See S1EA for Model I/III, EDAN for EDAS, and EDCE for Color Computer EDTASM+.

ALCW
Assembly-Language Coding, Ways to Generate Perfect Code

There are two standard ways to generate flawless assembly-language code:

1. Carefully design in detail by design specifications, flowcharts, careful coding, and structured programming techniques. Debugging should be minimal.

2. Go to a wastebasket and pick up any assembly-language listing. Use this listing and debug and modify it until it works the way the program is supposed to. Debugging will probably be no longer than method 1.

ALIB
Inserting a Line in BASIC, All Systems

BASIC will insert a new line in numerical order in a BASIC program in RAM if you simply type a new line with a line number that corresponds to the insertion point. For example, to insert a line between BASIC program line 100 and 112, simply type a new line with line number 111.

ALWI
Assembly Language, What is It?

Read MLWI, "Machine Language, What is It?"

An assembler program translates text representing machine language instructions into the actual machine language instructions. The text for the assembler is written in "assembly language."

Figure ALWI-1 shows a typical assembly language program "listing" or printed output of the assembler program for the Model I/III. Model II and Color Computer listings are similar.

Figure ALWI-1 – *Typical Assembly-Language Listing*

```
                 ;*********************************************************
                 Ø279Ø ;* PROCESS COMMAND LINE SUBROUTINE                          *
                 Ø28ØØ ;*     ENTRY:  (IX)=>FIRST CHAR OF COMMAND STRING            *
                 Ø281Ø ;*     EXIT:   (MARGN)=# OF ARGUMENTS FOUND, MAY BE Ø        *
                 Ø282Ø ;*             (MARG1)=>FIRST ARG STRING, TERM BY NULL       *
                 Ø283Ø ;*             (MARG1L)=LENGTH OF ARG                        *
                 Ø284Ø ;*             (MARG1D)=DELIMITER IN ASCII                   *
                 Ø285Ø ;*             (MARG2)=>SECOND ARG STRING, TERM BY NULL      *
                 Ø286Ø ;*             (MARG2L)=LENGTH OF ARG                        *
                 Ø287Ø ;*             (MARG2D)=DELIMITER IN ASCII                   *
                 Ø288Ø ;*********************************************************
A545 Ø625        Ø289Ø PCOMX  LD     B,37           ;37 BYTES TO CLEAR
A547 2143AD      Ø29ØØ        LD     HL,MARGN       ;AREA START
A54A AF          Ø291Ø        XOR    A              ;ZERO FOR FILL
A54B CD3FA5      Ø292Ø        CALL   FILLCH         ;ZERO ARGUMENTS
A54E Ø6ØA        Ø293Ø        LD     B,1Ø           ;FIND IN 1Ø
A55Ø CDØ6A5      Ø294Ø        CALL   FNBSTR         ;FIND NON-BLANK
A553 285F        Ø295Ø        JR     Z,PCOØ9Ø       ;GO IF NOT FOUND
A555 DDE5        Ø296Ø        PUSH   IX             ;SAVE START
A557 Ø6ØA        Ø297Ø        LD     B,1Ø           ;FIND IN 1Ø
A559 CDØ3A5      Ø298Ø        CALL   FNDSTR         ;FIND TERMINATOR
A55C E1          Ø299Ø        POP    HL             ;RESTORE START
A55D 2Ø52        Ø3ØØØ        JR     NZ,PCOØ8Ø      ;GO IF INVALID
A55F 3255AD      Ø3Ø1Ø        LD     (MARG1D),A     ;STORE DELIMITER
A562 78          Ø3Ø2Ø        LD     A,B            ;# OF CHARS BEFORE DELIM
A563 3254AD      Ø3Ø3Ø        LD     (MARG1L),A     ;SAVE LENGTH
A566 48          Ø3Ø4Ø        LD     C,B            ;# OF CHARS
A567 Ø6ØØ        Ø3Ø5Ø        LD     B,Ø            ;NOW IN BC
A569 1144AD      Ø3Ø6Ø        LD     DE,MARG1       ;DESTINATION
A56C B7          Ø3Ø7Ø        OR     A              ;TEST FOR Ø BYTES
A56D 28Ø4        Ø3Ø8Ø        JR     Z,PCOØ2Ø       ;GO IF Ø BYTES
A56F EDBØ        Ø3Ø9Ø        LDIR                  ;MOVE TO ARGUMENT VAR
A571 3EØ1        Ø31ØØ        LD     A,1
A573 3243AD      Ø311Ø PCOØ2Ø LD     (MARGN),A      ;ONE ARG TO THIS POINT
A576 3A55AD      Ø312Ø        LD     A,(MARG1D)     ;GET DELIMITER
A579 FEØD        Ø313Ø        CP     ENTER          ;IS IT END?
A57B 2837        Ø314Ø        JR     Z,PCOØ9Ø       ;DONE IF SO
A57D DD23        Ø315Ø        INC    IX             ;NOW DELIM+1
A57F Ø6ØA        Ø316Ø        LD     B,1Ø           ;FIND IN 1Ø
A581 CDØ6A5      Ø317Ø        CALL   FNBSTR         ;FIND NON-BLANK
A584 282B        Ø318Ø        JR     Z,PCOØ8Ø       ;GO IF ERROR
```

The text entered to the assembler is on the right and consists of four sections, usually in neat columns.

The second column of this section is the "op-code" mnemonic, an abbreviation for the machine language instruction. It's much easier to say "ADD," for example, then "Add Two 8-Bit Operands."

The third column contains the "operands" associated with the "op-code." The number and type of operands vary according to the machine-language instruction.

The first column is the optional "symbolic label" for the instruction and is equivalent to a BASIC line number. Symbolic labels are used so that a programmer doesn't have to keep recalculating memory addresses for the instruction – the assembler does this for him.

The last column contains optional comments.

The listing portion on the left is what the assembler program generates from the assembly language text. The extreme left column is the location at which each instruction resides, in hexadecimal. The next column contains the machine language code for the instruction, in hexadecimal. The next column is the edit line number for the assembly language text.

To use assembly language on the Model I and III, see S1EA or EDAN ; for the Color Computer, see EDCE.

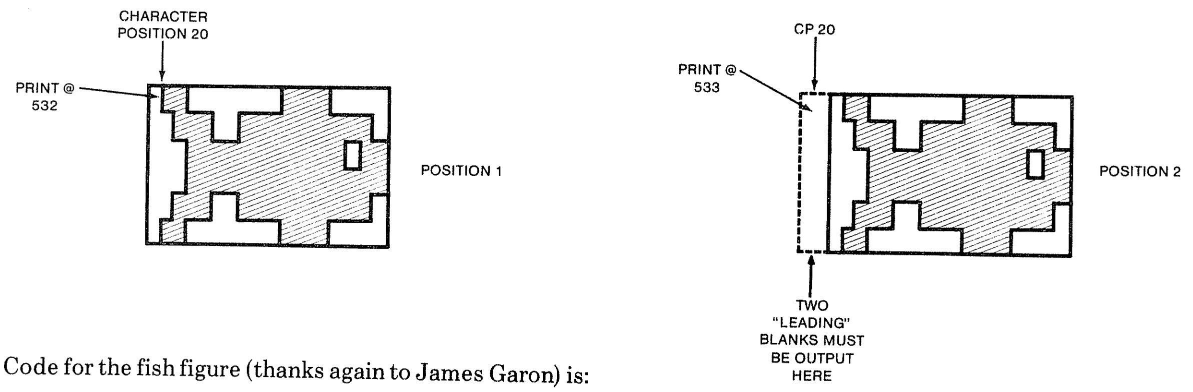
AN13
Animation, Model I and III

Read GHS1 for explanations of high-speed graphics. Normal SET/RESET graphics are not fast enough for effective animation.

The fastest graphics are done by assembly language and it may benefit you to learn assembly language, if you have more than a casual interest in graphics. (In fact, if you are very interested in graphics, get a Color Computer!) The POKE method explained in GHS1 can be used in assembly language for highest-speed graphics.

If you are working in BASIC, use either the POKE or string methods explained in GHS1 to move predefined figures. The best method is GHS1, Method 2, where a single string defines an entire figure. The figure can easily be moved around by using a moving starting position for the string and drawing the string at a series of locations, erasing any old pattern, as shown in Figure AN13-1.

Figure AN13-1 – *Animation Technique*



Code for the fish figure (thanks again to James Garon) is:

```
1ØØ ' FISH FIGURE
11Ø CLEAR 2ØØ
12Ø B$=CHR$(26)+STRING$(1Ø,24)
13Ø A$=" "+CHR$(13Ø)+CHR$(173)+CHR$(18Ø)+CHR$(184)
    +CHR$(188)+STRING$(2,191)+CHR$(156)+CHR$(18Ø)
    +B$+" "+CHR$(16Ø)+CHR$(158)+CHR$(135)+CHR$(139)
    +CHR$(143)+STRING$(2,191)+CHR$(143)+CHR$(135)
14Ø CLS
15Ø FOR P=64Ø TO 692
16Ø PRINT@P,A$:IF RND(2)=1 PRINT@P-54,"O";
17Ø FOR I=1 TO 3Ø:NEXT I
18Ø PRINT@P,CHR$(31)
19Ø PRINT@96Ø,:NEXT P
2ØØ GOTO 15Ø
```

AOIB
Arithmetic Operations in BASIC, All Systems

BASIC performs arithmetic (say air-ith-MATIC if you don't want to be shunned by Computer Science professors at TRS-80 cocktail parties) operations in about the same way you'd write down algebraic expressions.

Oh, oh . . . never written an algebraic expression, eh? BASIC uses variables named A, AA, GT, XC, BN, C1 or any two-character name, the first of which is an alphabetic character. You can actually use more than two characters, but BASIC will only look at the first two and it will get confused between variables such as ACCTSPAY and ACCTSRC.

Simplest operation:

```
100 AA=3.4
110 CX=56789
120 XX=34567.89999
```

sets variable AA equal to the value 3.4, variable CX to 56789, and variable XX to 34567.9. You can generally use any numeric value – integer, fraction, or mixed number. BASIC will take care of it automatically. There are special variables that you can use, but these are discussed in other topics. Use any variable names that follow the rules. You can use names as you think of them – they don't have to be predefined in general cases.

Addition and subtraction: Add or subtract by using a plus (+) or minus sign (-). You can generally make any number of additions and subtractions with impunity:

```
100 A=56.78-6-5.55-X+CV
```

Note that variables can be added and subtracted along with "constant" values. BASIC keeps a record of all variable values and simply uses the current value in the operations.

Multiplication and division: Use an asterisk (*) for multiplication and a slash(/) for division. Use them as often as you like. You may multiply constants or variables as before:

```
100 A=56*45.6*I         '56 times 45.6 times I
110 B=456/(56.6+JJ)  '456 divided by 56.6 plus JJ
```

Note in the above case that parentheses were used. Parentheses make certain that the addition of 56.6 plus variable JJ was done first, before the divide. See PHTU for a discussion of parentheses in expressions.

Exponentiation: To take a number to a power, use the up arrow. This is sometimes printed as a left bracket, depending upon the system.

```
100 A=B↑2             'variable B squared
110 A=B↑2.5           'oh yes, fractional powers are fine
```

Expressing Very Large or Very Small Numbers: Use the E format. In this format, a number is expressed similarly to "scientific notation" with a mixed number and power of 10. The power of 10 may be negative:

```
100 A=4.566E3         '4.566 times 1000
110 A=9.99E24         '9.99 times 10 to the 24th
120 B=.0078E-12       '.0078 times 10 to the -12th
```

Want to flex your programming muscles and try some of these operations? While in BASIC command mode, enter PRINT, followed by any expression, such as

```
PRINT 23.45*45.6+78
```

and BASIC will compute the value. You can't hurt anything, and BASIC will tell you if you exceed a number range or make an error. Sometimes you won't be able to decipher the error code, but it's all in this book.

See also "double-precision" formats in DPHU.

notes

BBUS
Bulletin Boards, Using

Data communication Bulletin Boards are message centers that you can dial up with your Model I, II, III or Color Computer system if you have an RS-232-C driver, the necessary hardware, and a modem (see MWAT, MHTU). At the current time I have a list of about 600 separate bulletin board systems. Although they started out being related to a specific type of computer, such as the TRS-80 Model I or Apple, most now service any type of computer. They all operate about the same way; most are free. They are located in every part of the country and all over the world, with the possible exception of North Dakota.

Most systems offer: capability of leaving any message (in good taste), reading messages, quick scanning of messages, and display of topical bulletins. Subject matter varies — requests for help, items for sale, software and hardware promotions, philosophical discussions, and sexual liasons. To "get on" a BBS, do the following:

1. Using the first part of procedure MHTU for a modem, or ACHT for a coupler, hook up your system to a modem.

2. Set your RS-232-C interface to word length of 8, 1 stop bits, no parity, 300 baud, full duplex.

3. Load a data terminal program. This can be a separate "stand alone" program or a program such as "LCOMM" in LDOS, Model I/III.

4. Complete procedure MHTU or ACHT to dial the BBS number and get a screen display.

5. At this point the BBS is usually very generous in prompting you about its operation. A typical dialogue is given in Figure BBUS-1. If you forget the commands, type H for "Help" or simply enter an invalid command; most systems will reply with a menu of valid commands.

Figure BBUS-1 – *Bulletin Board Dialogue*

```
AT DT 5377913
CONNECT
?????C/R ?2SHL.B.DO YOU NEED NULL'S (Y/N) ? N

CAN YOU RECEIVE LOWER CASE (Y/N) ? Y


Good Afternoon !

Welcome to the
Orange County TRS-8Ø Data System

What is your - FULL - name ? WILLIAM BARDEN

Your location (city,state) ? MISSION VIEJO, CA

Name: WILLIAM BARDEN
from: MISSION VIEJO, CA

Is this correct, WILLIAM? Y

Last message number in system is 1773

Logging WILLIAM BARDEN from MISSION VIEJO, CA to disk

You are caller No. 23243

Enter 'S' to skip!

     ********** B U L L E T I N S **********
-----------------------------------------------
 THE SYSTEM IS NOW RUNNING ON A TRS-8Ø MODEL III.

 ** DOWNLOAD AND UPLOAD IS NOW AVAILABLE **

          CHECK THE PROGRAM MENU!

 .................. E N D ..................

 Main System Menu
 ----------------
(S)ummary of messages in system        (E)nter a message
(Q)uick summary of messages            (K)ill a message
(H)elp with system operation           (L)og on again
(I)nformation about system             (O)ther system numbers
(P)rotected message retrieval          (R)etrieve messages
(B)ulletins
```

```
(A)SCII TEST
(C)LOCK (Time on line)
(T)ERMINATE CONNECTION
(M)ENU (transfer to programs)

<< SELECT >> ?S

Message summary
Do you wish selective summary ? N
```

BDSM
Backup Diskette, General

A backup diskette is generally a carbon copy of an original diskette. The original diskette must be a TRSDOS, LDOS, or other diskette that is capable of being backed up. Certain diskettes are in "non-standard" formats and cannot be backed up at all, or can be backed up only once or twice. Backups should be used in place of the original; the original should be used only to make backup copies for "working diskettes."

If you have a non-standard diskette that cannot be backed up or can be backed up only once or twice, refer to the software documentation for obtaining a backup or copy. This might involve ordering a second diskette from the software supplier. (Gee, I hope that the company who supplied your accounts receivable package is still around when your diskette is clobbered and you need a second copy. But then again, small businesses do not fail that often . . .).

If you have a standard operating system diskette — TRSDOS or LDOS, follow this procedure:

1. FORMAT the diskette. This involves writing a "skeleton" set of tracks and sectors on the diskette. The diskette comes from the factory essentially blank. The command for all standard operating systems is "FORMAT." Follow the procedure for the FORMAT command in the operating system documentation. In most cases, this will simply be entering "FORMAT" after the prompt, and answering the prompting questions about the drive number, password, etc. The FORMAT process takes about a minute.

2. BACKUP the diskette by the BACKUP command. This command makes an identical copy of the diskette, with the exception of the diskette name. The backup procedure takes about 2 minutes.

The backup copy can now be used the same way the original diskette was to be used.

BHTG
BASIC, How to Get to

Model I, non-disk: Power on (see TOCH) and RESET (RBWI). You should now see the message

```
MEMORY SIZE?
```

The MEMORY SIZE prompt asks whether you want to "protect" high memory (see PROT) for special applications. You don't need it at this point.

Press ENTER and you'll see

```
RADIO SHACK LEVEL II BASIC
READY
>_
```

You're now ready to start entering BASIC commands.

Model III, non-disk: Power on (TOCH) and RESET (RBWI). You should see the message

```
Cass?
```

This prompt lets you select a low or high cassette data rate (see FHBF). You don't need the low rate unless you're loading Model I compatible tapes or are just a masochist.

Press ENTER (selects high rate).

You'll now see the message

```
Memory Size?
```

Press ENTER and you'll see

```
Radio Shack Model III BASIC
(c) '8Ø Tandy
READY
>_
```

You're now ready to start entering BASIC lines.

Model I/III TRSDOS: Load TRSDOS by THT3. Now enter

```
BASIC
```

BASIC will load from disk (you'll hear it go "tink," and the red light will come on). You should then see

```
HOW MANY FILES?
```

Press ENTER. (Entering a value of 1 through 15 selects 1 to 15 file buffers, but simply ENTER gives you 3).

BASIC will then ask the question

```
MEMORY SIZE?
```

Press ENTER (no memory protected, see above or PROT).

BASIC will then print out a copyright message and end with

```
>_
```

You can now enter BASIC commands.

Model II TRSDOS: Power up by TOCH. After TRSDOS loads, you should see the impressive TRSDOS logo, followed by a prompt message for date. Enter the date by DSDS.

You should now see

```
TRSDOS READY
```

Set the CAPS key on the keyboard and enter

```
BASIC
```

in upper case. BASIC will load from disk (you'll hear it go "tink," and the red light will come on). You should then see the title BASIC message, ended by

```
>_
```

You're now ready to start entering BASIC commands. (To set more than 0 file buffers use

```
BASIC -F:n
```

where n is 1 through 15. You won't need disk buffers if you won't be writing your own *data* to disk, and that's unlikely if you're reading this).

Model I/III LDOS: Load LDOS by procedure L13L. Now enter

```
L BASIC
```

LBASIC (LDOS's BASIC) will load from disk (you'll hear it go "tink," and the red light will come on). You should then see the copyright message and

```
>_
```

You have not protected any memory and have 3 file buffers, but that's probably fine if you're reading this preliminary procedure. You're now ready to start entering BASIC commands.

BKHT
BREAK Key, How to Disable, Model I/III

Model I/III LDOS and Model III TRSDOS users, are you tired of your program operator stopping crucial bean-counting programs by inadvertently hitting BREAK? If you are, send $5 (no stamps) please, to TRS-80, Box CMD"B", Ripoff, TX for this ancient Egyptian secret.

Actually, save your money. Simply type

```
CMD"B","OFF"
```

to disable the BREAK key and

```
CMD "B","ON"
```

to enable the BREAK key.

When the BREAK key is disabled, it will only be active during cassette, printer or serial input/output, thus preventing the system from locking up completely. At the same time it'll be disabled during normal program execution.

The BREAK key is located off by itself at the right corner of the keyboard, but perhaps this command is useful for nearsighted users with negative thoughts or equal opportunity employers. (Think about it . . .).

Model I/III LDOS users: use the LDOS command

```
SYSTEM (BREAK=OFF)
```

and

```
SYSTEM (BREAK=ON)
```

to enable and disable the BREAK key outside of LBASIC programs.

BLTL
BASIC Line Too Long to Print or Display, Most Systems

(Not applicable to Model I/III, Level I or Color Computer).

Easy. Go to Edit mode (see EMBH). Find 50th character by entering 50 (space bar). Go to insert mode in Edit mode (see EMBH). Insert a down arrow. End insert mode (by SHIFT, up arrow). Find 50 more characters by 50 (space bar). If at end of line, you're done. If not, repeat the insert of the down arrow. Repeat until end of line. The BASIC listing will now not run "over the stops" on printing. Use other than the 50 spacing for screens or printers with fewer columns. Use spacing of other than 50 as applicable.

Here's an example:

The BASIC line before editing:

```
1ØØØ DA=31:IF (MO=8) AND (YR=1981) THEN DA=35
     ELSE IF MO=5 THEN DA=3Ø ELSE IF MO=7 THEN DA=3Ø
     ELSE IF MO=1Ø THEN DA=3Ø ELSE IF MO=12 THEN DA=3Ø
     ELSE IF MO=3 THEN DA=2 8+L
```

The BASIC line after editing:

```
1ØØØ DA=31:IF (MO=8) AND (YR=1981) THEN DA=35 ELSE
     IF MO=5 THEN DA=3Ø ELSE IF MO=7 THEN DA=3Ø ELSE
     IF MO=1Ø THEN DA=3Ø ELSE IF MO=12 THEN DA=3Ø ELSE
     IF MO=3 THEN DA=2 8+L
```

The down arrows won't display, but will cause a "carriage return" so that the line restarts.

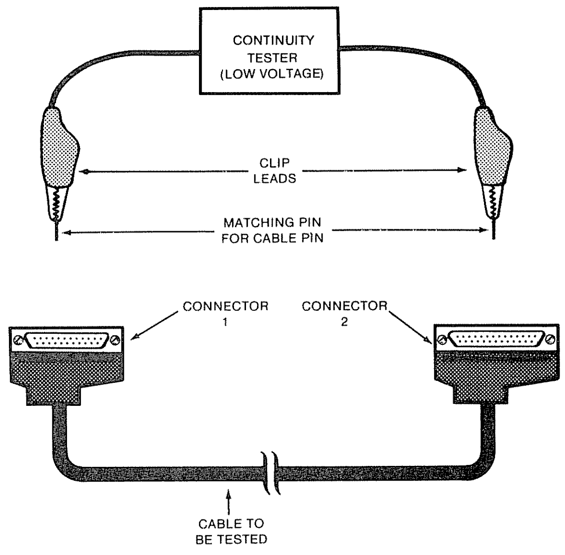
BOCH

Buzzing Out Cables, How to

The term "buzzing out" cables means that the cable connections from one end to the other are verified by completing an electrical circuit. The term undoubtedly comes from a buzzer being used as a continuity device, and not, as many think, from Milton J. Buzzinski, an early electrical technician.

The basic scheme is shown in Figure BOCH-1. It helps to have two people for this, although one person can do it with the help of electrical clips. Here's the procedure:

Figure BOCH-1 – *Buzz-Out Procedure*



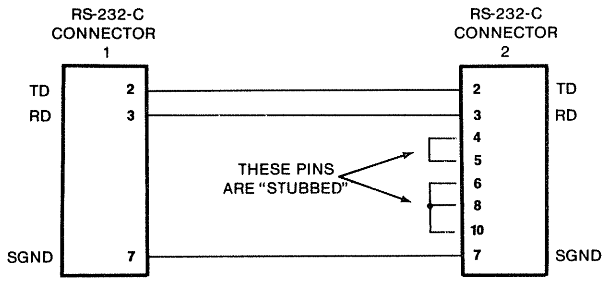
1. You must have a complete cable wiring table that defines which pin on one connector goes to a defined pin on the other connector. Some cables, such as RS-232-C cables, may also have pins that go to the same connector, as shown in Figure BOCH-2.

2. Go to your local Radio Shack store and pick up the following items:

A. Continuity tester. This is an inexpensive device (under $10) that indicates completion of an electrical circuit by a red LED (light emitting diode). There is absolutely no shock hazard and you can use the device with impunity.

An alternative to this device is a do-it-yourself continuity tester. This may be an LED/resistor combination, or simply a buzzer, as shown in Figure BOCH-3.

Figure BOCH-2 – *Typical RS-232-C Cable*



PIN LIST

CONNECTOR 1 PIN	CONNECTOR 2 PIN
2	2
3	3
7	7
—	4,5 CONNECTED
—	6,8,10 CONNECTED

Another alternative is a "voltmeter" (see VMHT). This is a versatile test instrument that will be invaluable for microcomputer use and also for household use.

B. Set of test leads with insulated clips on either end.

C. Now the hard-to-get part. If one or both of the plugs is a female (you know, the male has the protruding pins and the female - uh . . . you know what I mean, I'm sure . . .), then you'll need a pin to insert into each connector hole. This can be simply a piece of wire about the same size as the corresponding male pin, or an actual pin, "cannibalized" from a similar connector.

If one or both of the plugs is a male, then you'll need a corresponding female pin. These might be hard to find. One solution: Get a connector with matching female pins and break it apart.

3. You're now set to buzz out the cable. With the cable wiring table in front of you, methodically go down each row of pins, one at a time. Insert the wire or pin so that it

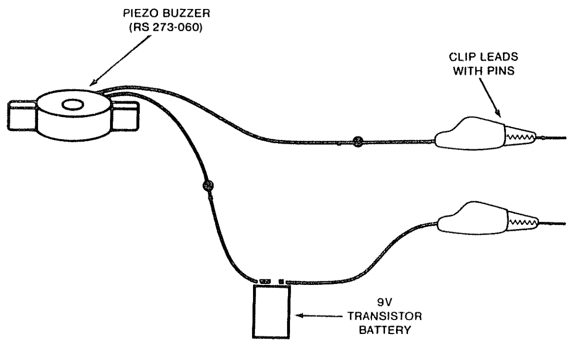
is solidly attached. Now find the corresponding pin on the other connector in the table or diagram and attach the other lead to the pin. You should hear a buzz, see the LED light, or see the multimeter scale show close to zero ohms to indicate continuity.

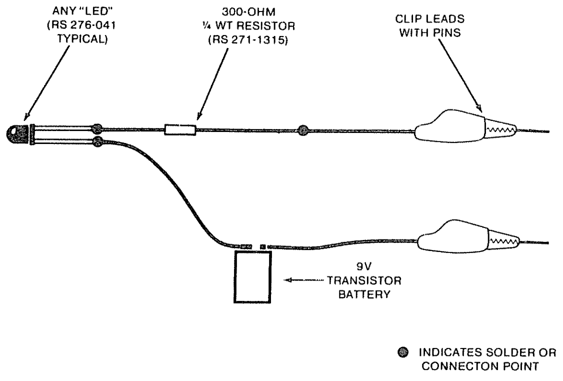
4. If you are using a multimeter, typical resistance readings should be a fraction of an ohm to 4 ohms, depending upon the cable length (see VMHT).

5. If you are unsure of the wiring job you did on the cable, also test continuity on both connectors for adjacent pins. For example, test between pins 1 and 2 and between pins 2 and 3 on the same cable; there should be no continuity, of course.

Don't hurry on this procedure, and be certain that you have the proper pins. If you suspect an "intermittent" cable, jiggle the cable slightly to see if you can break the continuity. Breaks will usually occur near the connector, unless the cable is unusually stressed.

Figure BOCH-3 – *Simple Continuity Tester*





BPFM
Breakpointing for Machine-Language Debugging, All Systems

Model I/III and Color Computer users: Read the procedure on the use of Model I/III DEBUG (DT1U) or Color Computer ZBUG (ZUEC) if you're unfamiliar with machine language debug packages.

The technique of breakpointing is a subset of the philosophy of debugging in general — bracket a problem area by a binary search. The bracketing procedure goes like this: Put a breakpoint at the furthest point in the code that you'll think you'll reach without a program "bomb". You may require several breakpoints, as there may be several divergent paths.

Execute from the last "good" instruction.

See which breakpoint, if any, you've reached. If you have reached a breakpoint, check the contents of registers, memory locations, and results for accuracy.

Repeat this technique for successive code. Eventually, you'll find a section of code in which the breakpoints are never reached. At this point, reload the code (things may be radically altered), put in new breakpoints halfway through the code, and try again. If the breakpoints are reached and results seem valid, breakpoint the last half and try again. If the breakpoints are not reached, breakpoint the first half.

Repeat the procedure until the point at which the blowup occurs has been found.

Isn't this too tedious? Yes, but common for assembly language debugging unless you're one of those gifted programmers who can generate code that runs the first time from your desk (and there are such people).

BPIB
Breakpointing in BASIC, All Systems

You can use the STOP command to break at any point in your BASIC program. Simply insert a STOP line at the point at which you wish to stop. If you want to STOP after line 300, for example, you'd have

```
3Ø0 A=B: C=IX(A,B)      'get array value
3Ø1 STOP                'this was inserted
31Ø PRINT "VALUE=";C    'print
```

At the STOP line, BASIC would display

```
BREAK IN 3Ø1
```

and you could then print out any variables by using a PRINT command in the command mode. (However, if you do any editing, those variables will be reset!).

STOP, then, is a handy way to check your progress through a BASIC program and see that things are going the way you expect them to.

After you've reached the STOP, enter a CONT command to continue from the point of the STOP. Pull out all the STOPs for the final program version.

Want to continue from another point with all variables intact? Enter GOTO nnnn to continue from line nnnn.

BPSP
Backup Procedure, Specific

Read BDSM for a general description of backups on diskettes, and then follow these procedures:

Model I, TRSDOS: Backup is in two steps, formatting the diskette and using the BACKUP command.

1. Format the diskette by entering

```
FORMAT
```

The FORMAT program will be loaded, and it will prompt you about the format. If you have a single drive system, you can backup from drive 0 by entering FORMAT and inserting the diskette to be formatted *after* the FORMAT program loads (disk light goes out).

Drive to be used: 0 or others. Diskette name: enter up to 8-character name. Creation date: today's date in date format. Master password: enter a 1 to 8 character name as "master password" (see PWDS). You don't want to lock out any tracks unless you have the ability to access the diskette directly, without TRSDOS. This involves using your own assembly-language driver, and few of us would use this option.

If the FORMAT procedure results in "locked-out" tracks, it means that the FORMAT process was unsuccessful in setting up these tracks. In this case try the FORMAT again. If the FORMAT fails again, throw away the diskette, rather than using locked-out tracks.

2. Backup the diskette by entering

```
BACKUP
```

After the BACKUP program loads (red light goes out), you can backup from a single disk drive (drive number 0), or from one drive to another. If using a single drive, answer 0 for source drive and 0 for destination drive. BACKUP will prompt you with messages about when to insert the source and destination diskettes. If you're using two drives, you won't have to swap diskettes.

Enter backup date as a standard system date.

3. You don't have to FORMAT the diskette if it is blank, but you do have to format it if it is not blank. If the disk is blank BACKUP will format automatically.

4. Believe it or not, you may have to erase a diskette using a "bulk tape eraser" (RS 44-232) to prepare it for formatting — what Harv Pennington calls a "pagan ritual." Follow the instructions on the eraser, you can't go wrong.

Model III, TRSDOS: It's not necessary to format the diskette before the backup.

1. Backup the diskette by entering

```
BACKUP
```

After the BACKUP program loads (red light goes out), you can backup from a single disk drive (drive number 0), or from one drive to another. If using a single drive, answer 0 for source drive and 0 for destination drive. BACKUP will prompt you with messages about when to insert the source and destination diskettes. If you're using two drives, you won't have to swap diskettes.

The diskette "master password" controls access to the diskette. Enter a 1-to 8-character name, and make certain you remember it (see FNMH).

If the diskette has been previously used, BACKUP will ask you if you want to use the diskette. This is a good check on whether you really want to destroy the diskette contents.

BACKUP will then ask if you want to reformat the diskette. This is usually a good idea, as the FORMAT "certifies" that the diskette can be written to and read from.

Formatting takes about 2 minutes, and the message "BACKUP COMPLETE" will terminate the process. There should be no "flawed" tracks; if there are, reformat and backup. You can use a diskette with flawed tracks, but I'd advise against it.

2. You can use the TRSDOS FORMAT program to prepare a data diskette (see DDWA). The procedure is identical to the FORMAT in BACKUP — you're asked for a name and master password, and a check is made on whether the diskette contains data.

Model II, TRSDOS: It's not necessary to format the diskette before the backup.

1. The simplest form of BACKUP is:

```
BACKUP
```

After the BACKUP program loads (red light goes out), you can backup from a single disk drive (drive number 0), or from one drive to another. If using a single drive, answer 0 for source drive and 0 for destination drive. BACKUP will prompt you with messages about when to insert the source and destination diskettes. If you're using two drives, you won't have to swap diskettes.

The diskette "master password" controls access to the diskette. Enter a 1- to 8-character name, and make certain you remember it (see FNMH).

If the diskette has been previously used, BACKUP will ask you if you want to use the diskette. This is a good check on whether you really want to destroy the diskette contents.

BACKUP will then ask if you want to reformat the diskette. This is usually a good idea, as the FORMAT "certifies" that the diskette can be written to and read from.

Formatting takes about 2 minutes, and the message "BACKUP COMPLETE" will terminate the process. There should be no "flawed" tracks; if there are, reformat and backup. You can use a diskette with flawed tracks, but I'd advise against it.

2. You can use the TRSDOS FORMAT program to prepare a data diskette (see DDWA). The procedure is identical to the FORMAT in BACKUP — you're asked for a name and master password, and a check is made on whether the diskette contains data.

3. There are a number of options that can be used with BACKUP. If no options are used, BACKUP asks you for information by "prompt messages." The source and destination master passwords and name of the new disk can be specified in the BACKUP command line by:

```
BACKUP Ø TO 1 PW=oldpassword NEW=newpassword ID=name
```

To copy system files only (and delete any user files), use the SYS option, along with any of the above:

```
BACKUP 1 TO Ø SYS
```

If you don't like to chat, the ABS option causes BACKUP to simply go ahead and make the backup, taking all of the information (password, name) from the source diskette.

```
BACKUP Ø TO 1 ABS
```

To copy selective files only, use the PROMPT option. BACKUP will ask you whether you want a file copied before the copy, and you can easily delete files by answering Y(es) or stop after a certain point by S(top). Note that *user files*, not system files, are involved.

NOAUTO does not copy any AUTO command (see AEDP). This is the same as backing up the diskette and then typing AUTO to stop the AUTO startup procedure.

4. The easiest FORMAT is with no options. To specify the disk name and password, use

```
FORMAT Ø ID=name PW=password
```

The ABS option specifies that no warning message will be displayed if the diskette contains data. It's a nicety to eliminate an annoying message for each format session:

```
FORMAT 1 ID=ACCOUNTS PASSWORD=BROKE ABS
```

TRSDOS generally has two directories (see DIDI), a primary and alternate directory. The alternate directory is used in lieu of the primary directory if there are disk errors associated with the primary. Try to avoid overused diskettes by doing periodic BACKUPs on a rotating basis. You'll find out how long you can use a diskette without errors primarily by experience (see DEHM). You *can* specify no alternate directory by

```
FORMAT 1 ALT=ØØ
```

but it's not advisable. You can also specify the location of the primary and alternate directory tracks (see DIDI). Normally the directories are at tracks 44 and 52, but you can move them based on your system requirements (you might have one large file and put the directories at the beginning of the diskette, for example). These options shouldn't be specified by the casual user. If you want to move the directories, keep a spread of 8 tracks between them and do something like

```
FORMAT 1 DIR=1 ALT 9
```

Model I/III, LDOS: Backup is in two steps, formatting the diskette and using the BACKUP command.

1. Format the diskette by entering

```
FORMAT
```

The FORMAT program will be loaded, and it will prompt you about the format. If you have a single-drive system, you can format from drive 0 by entering FORMAT and inserting the diskette to be formatted *after* the FORMAT program loads (disk light goes out).

Drive to be used: 0 or others. Diskette name: enter up to 8-character name. Master password: enter a 1- to 8-character name as "master password" (see PWDS).

For the remaining options (single or double density, sides, cylinders, bootstrap) simply hit ENTER if you aren't familiar with the options. This will invoke the "defaults" for your Radio Shack system, Model I or III. Defaults for the Model I are: single density, 1 side, 35 cylinders (tracks), and 40 millisecond stepping rate. Defaults for the Model III are: double density, 1 side, 40 cylinders (tracks), and 6 milliseconds stepping rate.

Non-standard disk drives: Use the SYSTEM command to reset the standard options for your drives before doing backups. You can specify 40 track and double-sided drives, 8-inch drives, or other configurations.

If the FORMAT procedure results in "locked out" tracks, it means that the FORMAT process was unsuccessful in setting up these tracks. In this case try the FORMAT again. If the FORMAT fails again, throw away the diskette, rather than using locked out tracks.

2. Backup the diskette in the simplest case by entering

```
BACKUP
```

After the BACKUP program loads (red light goes out), you can backup from a single disk drive (drive number 0), or from one drive to another. If using a single drive, answer 0 for source drive and 0 for destination drive. BACKUP will prompt you with messages about when to insert the source and destination diskettes. If you're using two drives, you won't have to swap diskettes.

3. Like many LDOS commands, there are a huge number of options for both FORMAT and BACKUP.

For FORMAT, specify NAME and MPW (master password) similarly to

```
FORMAT :1 (NAME=ACCOUNTS,MPW=BROKE)
```

The information relating to the disk drive configuration can be entered in a single FORMAT line by using SDEN or DDEN, SIDES=, CYL=, and STEP=.

Use QUERY=N and ABS to avoid prompt messages. This is useful for doing a FORMAT from JCL, although I can't visualize too many operations where a delayed format would be useful (how does the system *insert* the diskette? A hand comes out . . .).

For BACKUP, specify the source and destination drives in a single line by

```
BACKUP :Ø :1
```

You can copy VIS (visible files), SYS (system files), and/or INV (invisible) files by intermixing combinations of these options

```
BACKUP :1 :Ø (SYS,INV)
```

You can use the QUERY option to selectively copy files, answering Y(es) only to those files you want copied.

```
BACKUP :Ø :1 (QUERY)
```

You can use the OLD and NEW options to copy either only those files that already exist on the destination diskette, or those which do *not* exist:

```
BACKUP :Ø :1 (OLD)      (existing)
BACKUP :Ø :1 (NEW)      (only new)
```

You can use the DATE option to copy only files that fall between two dates. Dates recorded with the files are the dates of *last modification* of the file:

```
BACKUP :Ø :1 (DATE="11/Ø1/82-11/15/82")
```

You can use the MOD option to transfer only those files that have been modified since the last backup:

```
BACKUP :Ø :1 (MOD)
```

The X option allows you to backup a data diskette without having a system disk in a single-drive system. BACKUP will prompt you to insert the source, destination, or system diskettes, and you'll need at least three hands, but it's better than some other DOSes I could mention.

You can use a "wild card" character to backup only those files that meet the requirements. The wild card character takes the place of a string of characters. To back up only those files with the file extension /FEB, for example, use:

```
BACKUP $/FEB:Ø :1
```

Using /FEB:0 :1 has the same effect.

To backup all 6-character file names that start with the the characters "ACCT", do:

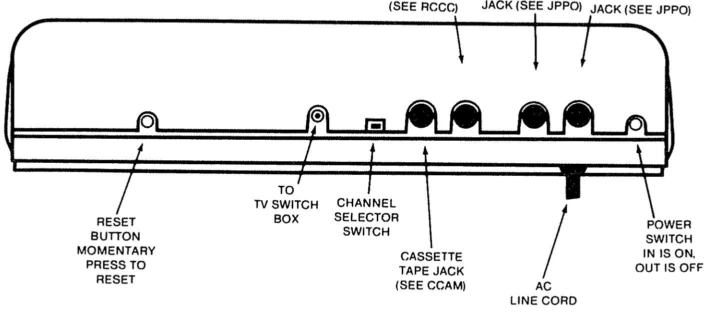
```
BACKUP ACCT$$/$:Ø :1
```

Generally, you can use the options above in any combination. You can also abbreviate by Q, I, M, S, V, and D.
More than you ever wanted to know about BACKUP . . .

BSCC
Switches, and Connectors, Color Computer

Refer to Figure BSCC-1. All of the switches and connectors are on the rear of the CC except for the ROM cartridge connector, which is in a spring-loaded door on the right-hand side of the cabinet. The "pin-outs" of the connectors are detailed as indicated in the figure. The pin-out of the ROM cartridge connector is given in RCCN.

Figure BSCC-1 – *Color Computer Switches and Connectors*


RS-232-C JACK (SEE RCCC)
RIGHT JOYSTICK JACK (SEE JPPO)
LEFT JOYSTICK JACK (SEE JPPO)
RESET BUTTON MOMENTARY PRESS TO RESET
TO TV SWITCH BOX
CHANNEL SELECTOR SWITCH
CASSETTE TAPE JACK (SEE CCAM)
AC LINE CORD
POWER SWITCH IN IS ON, OUT IS OFF

BSEC
To Backspace and Erase Character

All systems except Model II: Left arrow.

Model II: BACKSPACE.

BSEL
To Backspace and Erase Line

Models I/III and Color Computer: SHIFT, left arrow pressed simultaneously.

BSER
BS Error

Bad subscript. You've made a reference to an array with a subscript that was less than 0 or beyond the maximum you used in the DIMension statement, as in

```
100 DIM IX(100,2)     '101 by 3 array
   .
   .
1000 A=IX(56,3)       '2nd subscript must be 0, 1, or 2
```

notes

C1T3
Converting Model I TRSDOS Files to Model III TRSDOS Files

If you have a Model III and want to convert Model I disk files, you'll have to "convert" the Model I files to Model III files by using the Model III CONVERT utility program. You'll need two disk drives. This program can read a Model I diskette, find a file, and then copy the file to the Model III diskette; Model III TRSDOS by itself can't read a Model I diskette because of disk format differences.

To convert the files from the Model I diskette to files on a Model III diskette, do the following:

1. (Optional). Remove all passwords from the Model I diskette by using the ATTRIB command on the Model I system (ADFC). Don't have a Model I system? No problem, continue.

2. Prepare a Model III system diskette with enough room to receive the Model I files by deleting user files (see DFDA). Check the file space available by DSHL.

3. "Boot up" the Model III diskette to get TRSDOS READY. You now have the Model III diskette in drive 0.

4. Load CONVERT by entering

```
CONVERT
```

5. Load the Model I diskette in another drive, and answer the "SOURCE DRIVE?" prompt with the drive number — 1, 2 or 3.

6. Answer the "DESTINATION DRIVE" question with 0.

7. CONVERT will now look at each non-system file and copy it from the Model I diskette to the Model III diskette. If the same file name is on the Model III diskette, CONVERT will ask "FILE EXISTS, USE IT?". Answer Y or N. As each file is converted, the file name will be displayed on the screen.

8. If the Model I file is protected by a password, CONVERT will ask for the password. Enter the password.

Don't know the password? If you don't know the individual file passwords but do know the disk master password (try "PASSWORD"), see step 1 above. Don't know the master or individual passwords? Sorry, you can't convert the file on TRSDOS.

9. CONVERT will stop if it runs out of room on the Model III disk with a disk full error. In this case, all of the files copied up to this point on the Model III diskette are all right. Repeat the procedure with a new Model III diskette.

Be aware that the copied files may not necessarily run properly on the Model III because of "architectural" differences between the two machines. You probably won't be able to run Model I Visischlock on the Model III, for example, but Model I BASIC files that don't do anything fancy with "embedded" assembly language will probably be all right. See Radio Shack's "Instructions for Converting Specified Model I Programs for Use on TRS-80 Model III."

CBBH
Converting Between Binary and Hexadecimal

Easy. To convert from binary to hexadecimal, group the binary number into groups of 4 binary digits, starting from the right. Convert each group to a single hexadecimal digit (0000 is **0**, 0001 is **1**, 0010 is **2**, 0011 is **3**, 0100 is **4**, 0101 is **5**, 0110 is **6**, 0111 is **7**, 1000 is **8**, 1001 is **9**, 1010 is **A**, 1011 is **B**, 1100 is **C**, 1101 is **D**, 1110 is **E** and 1111 is **F**).

See example in Figure CBBH-1.

To convert from hexadecimal to binary, convert each hexadecimal digit to its binary equivalent of 4 bits as in the paragraph above. See example in CBBH-2.

Figure CBBH-1 – *Converting from Binary to Hexadecimal*

```
                CONVERTING FROM BINARY TO HEXADECIMAL

STEP 1: GROUP BITS

                 0010^0111^1011^0101

STEP 2: CONVERT EACH GROUP TO A HEX DIGIT

               0010    0111    1011    0101
                 ↓       ↓       ↓       ↓
                 2       7       B       5
```

$00\ 10\ 01\ 11\ 10\ 11\ 01\ 01_2 = 27B5_{16} = 10165_{10}$

Figure CBBH-2 – *Converting from Hexadecimal to Binary*

```
            CONVERTING FROM HEXADECIMAL TO BINARY

    STEP 1: CONVERT EACH HEX DIGIT TO A 4-BIT BINARY GROUP

                 5F29
                   ⇓
            5        F        2        9
            ↓        ↓        ↓        ↓
          Ø1Ø1     1111     ØØ1Ø     1ØØ1

    STEP 2: MERGE GROUPS
```

$Ø1\ Ø1\ 11\ 11\ ØØ\ 1Ø\ 1Ø\ Ø1_2 = 5F29_{16} = 24361_{1Ø}$

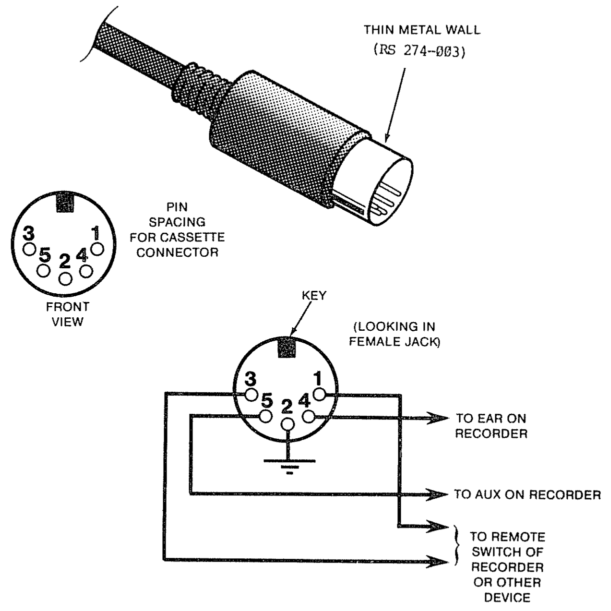
CCAM
Cassette Connector, Model I/III/Color Computer

The cassette connector on the Model I, III, and Color Computer has the same pin numbering and functions. Use the "thin-wall" (metal) version of a standard 5-pin DIN male audio plug to fit any cassette connector (Radio Shack 274-003).

Refer to Figure CCAM-1. Pins 1 and 3 connect to an internal relay and can be used as a programmable switch for low voltage, low current applications. Do not use for over 12 volts dc and 1/2 ampere or so. If you connect a milling machine to these pins, *you* will be responsible for the results. These pins normally control the cassette REMote input to turn the recorder on and off.

Pin 2 is signal ground. Pin 4 is the input to the computer from the EARphone jack of the recorder. Pin 5 is the output from the computer to the AUXiliary input of the recorder. See CRPI for a description of cassette recorder plugs.

Figure CCAM-1 – *Cassette Connector Pins*

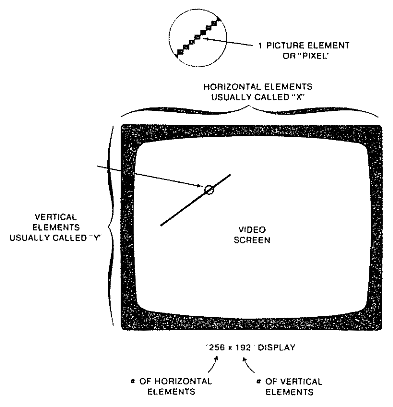


CCPA
Color Computer Pixel Addressing

All graphics commands in Extended Color BASIC use the highest resolution mode to specify the x,y coordinates. In this mode (256 by 192), x may be 0 through 255 or y may be 0 through 191. These values define the smallest graphics element that may be displayed, one pixel, named after Herman Pixel, who did early work in PICture ELements. (Each character position is 8 pixels by 12 pixels). Each element may be 1 by 1 pixel to 4 by 6 pixels, depending upon the mode. See Figure CCPA-1.

Don't worry about the fact that the resolution may be too coarse to pinpoint element 128,96, just use those values with impunity.

Figure CCPA-1 – *Display Nomenclature*


1 PICTURE ELEMENT OR "PIXEL"
HORIZONTAL ELEMENTS USUALLY CALLED "X"
VERTICAL ELEMENTS USUALLY CALLED "Y"
VIDEO SCREEN
"256 x 192" DISPLAY
# OF HORIZONTAL ELEMENTS
# OF VERTICAL ELEMENTS

CD13
Cassette DEBUG, Models I and III

TBUG was an early cassette-based DEBUG package for the Model I and III. I would strongly advise putting TBUG in the trash compacter and buying a copy of cassette-based DEBUG for the Model I/III. This DEBUG package is similar to Model III DEBUG (see DT1U) and will work in Level I or Level II/III.

The D, X, S, semicolon, minus, M, R, J, I, C, U and Q commands work identically to Model III DEBUG, and you can read the material in DT1U to see how they function. (I, C, the breakpoint option of J, and the T entry point and name options of W do not work on Level I, however).

Additional commands are T (Load a SYSTEM tape) and W (Write a SYSTEM tape).

The W command allows you to dump memory to cassette tape as a SYSTEM file (see LMFN). The resulting file can be loaded in a a BASIC SYSTEM command or by DEBUG using the L command.

Entering W will cause DEBUG to ask for the S(tart) address, the E(nd) address, the T(ransfer) address and the N(ame) of the file. Addresses should be in hexadecimal (see CFDH). Press BREAK to cancel on name entry or ENTER with no address for S, E, or T, (but only if the moon is in half phase).

Entering T will load the next file from cassette. If you're using a Model III, location **4211H** should be set to a 0 for 500 baud, or 1 for 1500 baud. My recommendation: always use 1500 baud. DEBUG can be used to M(odify) location **4211H** to the proper value.

Restart point for DEBUG: **4909H**

Load DEBUG as a SYSTEM tape with file name DEBUG. See LMFN for loading instructions.

CDFH
CREATEing a Disk File, How to Model I/II/III

The TRSDOS CREATE command can be used to create a "preallocated" file. The file space that you specify is set aside and it is "tagged" with the file name that you've specified in the CREATE command. Normally, TRSDOS allocates the space as it requires it, one segment at a time. Why use CREATE?

One very good reason is that the CREATEd file space, if done on a close-to-empty diskette, will be a "contiguous" area, and that will speed up disk accesses (see DFAH). Another reason is that you may know exactly how much space is required, and simply want to set aside that space to make certain you don't run out of room when you start using the disk.

To CREATE file space, you must know about how many records you'll be using and how large the records will be. Try to approximate the space required. If you do go over that amount, TRSDOS will allocate a new area on the disk, but it may not be contiguous.

Model III TRSDOS: Now use the CREATE command as follows:

```
CREATE name (REC=nnn,LRL=mmm)
```

where name is a standard disk file name (see FNMH), the REC value is the number of records required, and the LRL value is the Logical Record Length (0=256 bytes). Both the REC and LRL values are decimal values. The REC times the LRL is the number of bytes allocated, although this figure may be modified to the next largest granule (see DSHL).

Model II TRSDOS: The simplest form of CREATE is

```
CREATE name NREC=nnn,LRL=mmm
```

where name is a standard disk file name (see FNMH), the NREC value is the number of records required, and the LRL value is the Logical Record Length (0=256 bytes). Both the NREC and LRL values are decimal values. The NREC times the LRL is the number of bytes allocated, although this figure may be modified to the next largest granule (see DSHL).

An alternative way to allocate is to use the number of granules required. One granule on the Model II is 1280 bytes. Use

```
CREATE name NGRANS=nnn
```

where nnn is a decimal value representing the number of granules required. (Divide the number of bytes required by 1280).

Use the TYPE option of CREATE to indicate whether the file is going to be a fixed-length type or variable-length type. The "default" is fixed length:

```
CREATE ACCTS/FEB GRAN=25,TYPE=V
CREATE ACCTS/FEB GRAN=25,TYPE=F
```

Model I/III LDOS: Use the same format as Model III TRSDOS:

```
CREATE name (REC=nnn,LRL=mmm)
```

or use an alternative format that specifies the size in K byte *blocks* required

```
CREATE name (SIZE=nnn)
```

where nnn is the number of blocks of 1024 bytes required. For example, if you required 50,000 bytes for 500 records of 100 bytes logical record length, you'd have:

```
CREATE name (SIZE=49)
```

giving you 50,176 bytes in the file.

CDFL
Copying Disk Files, Model I/III LDOS

Tch, tch! These people at LDOS have stayed up nights thinking of options for their commands . . . (thank goodness).

To copy one disk file to another diskette, two or more drives:

```
COPY file1 TO file2
```

The "file1" and "file2" names are standard file names (see FNMH). If the file name exists on the destination disk, the file will be overwritten.

The TO is optional.

To copy one disk file to another diskette, single drive systems:

```
COPY file1:Ø TO file2:Ø
```

The "file1" and "file2" names are standard file names (see FNMH). If the file name exists on the destination disk, it will be overwritten.

Both diskettes should be LDOS system diskettes.

This copy can also be used to replicate a file on the same diskette. LDOS will prompt you to swap diskettes, but simply hit ENTER for each prompt.

To copy a disk file to another diskette without an LDOS system on either diskette (data diskette file to data diskette file or other Operating System to other Operating System diskette):

```
COPY file1:n (X)
```

The :n parameter here is the disk drive number, which may be any drive on your system. LDOS will prompt you to switch diskettes. There are three diskettes involved, and you must not get them mixed up! There's the source diskette and destination diskette, and the third is the "system" diskette containing the LDOS system. Doing the copy involves juggling the three diskettes and inserting the proper one when prompted.

Specifying a new logical record length for the destination file:

The optional LRL parameter lets you change the logical record length of the destination file. Without the LRL parameter, the LRL in the destination directory will be set to the same length as in the source directory. The LRL may need to be changed to make the file compatible with a program that expects a certain LRL.

To change the LRL, use one of the above formats and do:

```
COPY file1 TO file2 (LRL=nnn)
```

where nnn is the new logical record length of 0 (256) through 255.

Duplicating the attributes of the file when COPYing:

The optional CLONE parameter allows you to do a copy and also copy the visibility (see VAIF), the access and update passwords (FNMH), the protection level (ADFC, ADFL), and the LDOS "create" and "modified" status flags.

Without CLONE, the attributes of the destination file remain unchanged if it was "copied over," except for date. If the file is a new file, the attributes are changed to visible and both passwords are changed to the source file update password. With CLONE, the source file attributes are carried over. Use any of the above formats and CLONE:

```
COPY file1 TO file2 (CLONE)
```

Notes:

1. The LRL, CLONE, and X options may be used in any combination.

2. Partial file specifications can be used as in COPY ACCTS:0 :1 and COPY ACCTS/DAT:1 TO /DAT:0.

3. An extension of "/" as in COPY ACCTS/DAT:0 TO ACCTS/:1 signifies no extension.

CDHT
Clock Display, How to Use, TRSDOS/LDOS, Model I/II/III

To turn the screen display of the real-time clock on or off, enter

```
CLOCK (ON)        (Model I/III)
CLOCK ON          (Model II)
```

or

```
CLOCK (OFF)       (Model I/III)
CLOCK OFF         (Model II)
```

The real-time-clock is always running, except during cassette and disk I/O (see RTCN); the TRSDOS or LDOS CLOCK command simply enables or disables the display. To set the clock, see the TIME command (TSRT).

Model III, Level III: RAM locations 16919-16924 contain 6 values that define the date and time:

```
16919 = seconds
   2Ø = minutes
   21 = hours
   22 = years
   23 = days
   24 = months
```

These locations are set to 0 on computer start up, and are updated continuously. POKE the proper values to set the time and date and you can use the TIME$ BASIC command to get the current time in BASIC (see TIBP).

CDOC
Changing the Display Offset in the Color Computer

The display offset in the SAM in the Color Computer (see GPAR) determines which part of RAM will be displayed on the screen. If an alphanumeric mode is in force (by V2=V1=V0=0), then the display will be the 512 bytes of a text page. If a graphics mode is in force, then the display will be the appropriate number of bytes of the graphic page. The BASIC interpreter stores the proper address in the SAM depending upon the SCREEN command, the graphics page selected by PMODE and the PMODE (see SUCG).

You can select any memory starting address on 512-byte boundaries (**$0000**, **$0200**, **$0400**, etc) by POKEing into locations **$FFC6** through **$FFD3**, as shown in Figure CDOC-1. If you do this in the text mode, you can see any area of RAM or ROM displayed in color. The most interesting area is in page 0 (**$0000**), which shows the changing working variables in BASIC. Color debugging!

This starting address can be changed dynamically to display different graphics areas even if you do not have Extended Color BASIC.

Figure CDOC-1 – *Video RAM Starting Address*

POKE ADDRESSES							STARTING DISPLAY ADDRESS	
FFD3,2	FFD1,Ø	FFCF,E	FFCD,C	FFCB,A	FFC9,8	FFC7,6		
Ø	Ø	Ø	Ø	Ø	Ø	Ø	$ØØØØ	Ø
Ø	Ø	Ø	Ø	Ø	Ø	1	$Ø2ØØ	512
Ø	Ø	Ø	Ø	Ø	1	Ø	$Ø4ØØ	1Ø24
Ø	Ø	Ø	Ø	Ø	1	1	$Ø6ØØ	1536
Ø	Ø	Ø	Ø	1	Ø	Ø	$Ø8ØØ	2Ø48
			↓					
1	1	1	1	1	1	1	$FEØØ	65Ø24

CFAD
Copying a File To Another Diskette, Model I/II/III TRSDOS, Color Computer

If you have more than one drive, use the COPY command to copy the file as follows:

```
COPY FILE1:Ø TO FILE2:1
```

or

```
COPY FILE1:Ø :1
```

The filenames can have the standard format described in FNMH. TRSDOS will copy the file without any human interaction. The "source" and "destination" drives may be any drive numbers, 0 through 3. They may even be the same drive number (see CFSD).

(Not applicable to Model I): If you have a single drive, use the COPY command as follows:

```
COPY FILE1:Ø TO FILE2:Ø
```

The file names can have the standard format described in FNMH. TRSDOS will prompt you to insert "SOURCE" disk and insert "DESTINATION" disk at the appropriate times.

CFBD
Converting From Binary to Decimal

If you have a binary number that you would like to convert to decimal, do the following:

1. If the binary number is 0 through 1111111111 use Table CFDB-1.

2. If the number is greater than 1111111111, do the following:

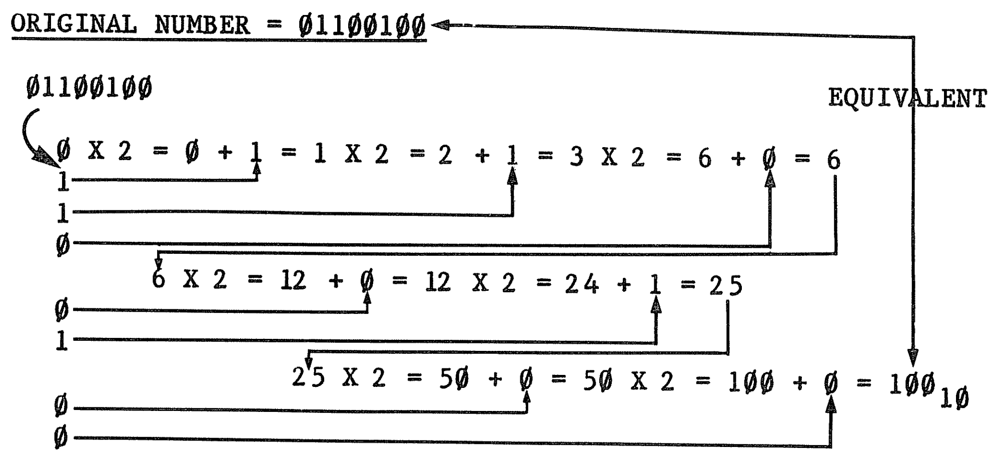
   A. Starting with the leftmost binary digit, multiply by 2. Add the 0 or 1 of the *next* binary digit.

   B. Multiply this result by 2 and add to the next binary digit.

   C. Repeat step B until the rightmost binary digit has been added. The result is the equivalent decimal number.

As an example: Convert 01100100 to decimal. See Figure CFBD-1.

Figure CFBD-1 – *Converting from Binary to Decimal*


ORIGINAL NUMBER = Ø11ØØ1ØØ
Ø11ØØ1ØØ
EQUIVALENT
Ø X 2 = Ø + 1 = 1 X 2 = 2 + 1 = 3 X 2 = 6 + Ø = 6
6 X 2 = 12 + Ø = 12 X 2 = 24 + 1 = 25
25 X 2 = 5Ø + Ø = 5Ø X 2 = 1ØØ + Ø = 1ØØ10

CFCS
Converting the First Character of a String to Numeric in BASIC, Most Systems

(Not applicable to Model I/III Level I).

The ASC function converts the first character of a string to numeric. In this code:

```
1ØØ A$="A WOMAN IS A WOMAN BUT A TRS-8Ø..."
11Ø NM=ASC(A$)
```

the result in numeric variable NM is a decimal 65 (**41H**), which represents the ASCII code of A (see ADFW for ASCII codes).

Why have this function? Why not? ASCII codes, fortunately, have the same "ascending sequence" order as A through Z, a through z, and 0 through 9, making this command useful in alphabetizing and other processing. ASC is the "inverse" of CHR$ (CUSE).

CFDB
Converting from Decimal to Binary

If you have a decimal number you want to convert to a binary value, do the following:

1. If the decimal number is 0 to 1023, use Table CFDB-1.

2. If the number is greater than 1023 (or equal to or less than 1023 and you feel like some math) do this:

   A. Divide the number by 2. Save the remainder as R1.

   B. Divide the result of step A by 2 again. Save the remainder as R2.

   C. Repeat step B until the amount remaining is 0.

   D. Arrange the remainders in reverse order. The result is the equivalent binary number.

As an example: Convert 100 decimal to binary. See Figure CFDB-1 for the process.

Figure CFDB-1 – *Converting from Decimal to Binary*

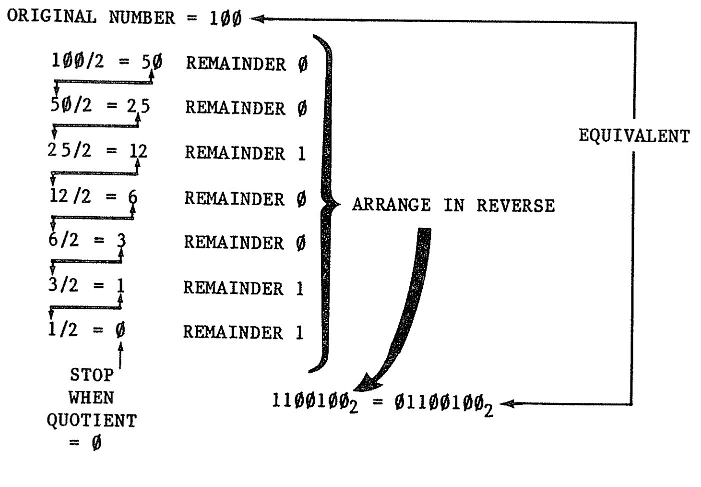


Table CFDB-1 – *Binary, Decimal, Hexadecimal*

DEC	BINARY	HEX	DEC	BINARY	HEX	DEC	BINARY	HEX
Ø	ØØØØØØØØØØ	ØØØ	53	ØØØØ11Ø1Ø1	Ø35	1Ø6	ØØØ11Ø1Ø1Ø	Ø6A
1	ØØØØØØØØØ1	ØØ1	54	ØØØØ11Ø11Ø	Ø36	1Ø7	ØØØ11Ø1Ø11	Ø6B
2	ØØØØØØØØ1Ø	ØØ2	55	ØØØØ11Ø111	Ø37	1Ø8	ØØØ11Ø11ØØ	Ø6C
3	ØØØØØØØØ11	ØØ3	56	ØØØØ111ØØØ	Ø38	1Ø9	ØØØ11Ø11Ø1	Ø6D
4	ØØØØØØØ1ØØ	ØØ4	57	ØØØØ111ØØ1	Ø39	11Ø	ØØØ11Ø111Ø	Ø6E
5	ØØØØØØØ1Ø1	ØØ5	58	ØØØØ111Ø1Ø	Ø3A	111	ØØØ11Ø1111	Ø6F
6	ØØØØØØØ11Ø	ØØ6	59	ØØØØ111Ø11	Ø3B	112	ØØØ111ØØØØ	Ø7Ø
7	ØØØØØØØ111	ØØ7	6Ø	ØØØØ1111ØØ	Ø3C	113	ØØØ111ØØØ1	Ø71
8	ØØØØØØ1ØØØ	ØØ8	61	ØØØØ1111Ø1	Ø3D	114	ØØØ111ØØ1Ø	Ø72
9	ØØØØØØ1ØØ1	ØØ9	62	ØØØØ11111Ø	Ø3E	115	ØØØ111ØØ11	Ø73
1Ø	ØØØØØØ1Ø1Ø	ØØA	63	ØØØØ111111	Ø3F	116	ØØØ111Ø1ØØ	Ø74
11	ØØØØØØ1Ø11	ØØB	64	ØØØ1ØØØØØØ	Ø4Ø	117	ØØØ111Ø1Ø1	Ø75
12	ØØØØØØ11ØØ	ØØC	65	ØØØ1ØØØØØ1	Ø41	118	ØØØ111Ø11Ø	Ø76
13	ØØØØØØ11Ø1	ØØD	66	ØØØ1ØØØØ1Ø	Ø42	119	ØØØ111Ø111	Ø77
14	ØØØØØØ111Ø	ØØE	67	ØØØ1ØØØØ11	Ø43	12Ø	ØØØ1111ØØØ	Ø78
15	ØØØØØØ1111	ØØF	68	ØØØ1ØØØ1ØØ	Ø44	121	ØØØ1111ØØ1	Ø79
16	ØØØØØ1ØØØØ	Ø1Ø	69	ØØØ1ØØØ1Ø1	Ø45	122	ØØØ1111Ø1Ø	Ø7A
17	ØØØØØ1ØØØ1	Ø11	7Ø	ØØØ1ØØØ11Ø	Ø46	123	ØØØ1111Ø11	Ø7B
18	ØØØØØ1ØØ1Ø	Ø12	71	ØØØ1ØØØ111	Ø47	124	ØØØ11111ØØ	Ø7C
19	ØØØØØ1ØØ11	Ø13	72	ØØØ1ØØ1ØØØ	Ø48	125	ØØØ11111Ø1	Ø7D
2Ø	ØØØØØ1Ø1ØØ	Ø14	73	ØØØ1ØØ1ØØ1	Ø49	126	ØØØ111111Ø	Ø7E
21	ØØØØØ1Ø1Ø1	Ø15	74	ØØØ1ØØ1Ø1Ø	Ø4A	127	ØØØ1111111	Ø7F
22	ØØØØØ1Ø11Ø	Ø16	75	ØØØ1ØØ1Ø11	Ø4B	128	ØØ1ØØØØØØØ	Ø8Ø
23	ØØØØØ1Ø111	Ø17	76	ØØØ1ØØ11ØØ	Ø4C	129	ØØ1ØØØØØØ1	Ø81
24	ØØØØØ11ØØØ	Ø18	77	ØØØ1ØØ11Ø1	Ø4D	13Ø	ØØ1ØØØØØ1Ø	Ø82
25	ØØØØØ11ØØ1	Ø19	78	ØØØ1ØØ111Ø	Ø4E	131	ØØ1ØØØØØ11	Ø83
26	ØØØØØ11Ø1Ø	Ø1A	79	ØØØ1ØØ1111	Ø4F	132	ØØ1ØØØØ1ØØ	Ø84
27	ØØØØØ11Ø11	Ø1B	8Ø	ØØØ1Ø1ØØØØ	Ø5Ø	133	ØØ1ØØØØ1Ø1	Ø85
28	ØØØØØ111ØØ	Ø1C	81	ØØØ1Ø1ØØØ1	Ø51	134	ØØ1ØØØØ11Ø	Ø86
29	ØØØØØ111Ø1	Ø1D	82	ØØØ1Ø1ØØ1Ø	Ø52	135	ØØ1ØØØØ111	Ø87
3Ø	ØØØØØ1111Ø	Ø1E	83	ØØØ1Ø1ØØ11	Ø53	136	ØØ1ØØØ1ØØØ	Ø88
31	ØØØØØ11111	Ø1F	84	ØØØ1Ø1Ø1ØØ	Ø54	137	ØØ1ØØØ1ØØ1	Ø89
32	ØØØØ1ØØØØØ	Ø2Ø	85	ØØØ1Ø1Ø1Ø1	Ø55	138	ØØ1ØØØ1Ø1Ø	Ø8A
33	ØØØØ1ØØØØ1	Ø21	86	ØØØ1Ø1Ø11Ø	Ø56	139	ØØ1ØØØ1Ø11	Ø8B
34	ØØØØ1ØØØ1Ø	Ø22	87	ØØØ1Ø1Ø111	Ø57	14Ø	ØØ1ØØØ11ØØ	Ø8C
35	ØØØØ1ØØØ11	Ø23	88	ØØØ1Ø11ØØØ	Ø58	141	ØØ1ØØØ11Ø1	Ø8D
36	ØØØØ1ØØ1ØØ	Ø24	89	ØØØ1Ø11ØØ1	Ø59	142	ØØ1ØØØ111Ø	Ø8E
37	ØØØØ1ØØ1Ø1	Ø25	9Ø	ØØØ1Ø11Ø1Ø	Ø5A	143	ØØ1ØØØ1111	Ø8F
38	ØØØØ1ØØ11Ø	Ø26	91	ØØØ1Ø11Ø11	Ø5B	144	ØØ1ØØ1ØØØØ	Ø9Ø
39	ØØØØ1ØØ111	Ø27	92	ØØØ1Ø111ØØ	Ø5C	145	ØØ1ØØ1ØØØ1	Ø91
4Ø	ØØØØ1Ø1ØØØ	Ø28	93	ØØØ1Ø111Ø1	Ø5D	146	ØØ1ØØ1ØØ1Ø	Ø92
41	ØØØØ1Ø1ØØ1	Ø29	94	ØØØ1Ø1111Ø	Ø5E	147	ØØ1ØØ1ØØ11	Ø93
42	ØØØØ1Ø1Ø1Ø	Ø2A	95	ØØØ1Ø11111	Ø5F	148	ØØ1ØØ1Ø1ØØ	Ø94
43	ØØØØ1Ø1Ø11	Ø2B	96	ØØØ11ØØØØØ	Ø6Ø	149	ØØ1ØØ1Ø1Ø1	Ø95
44	ØØØØ1Ø11ØØ	Ø2C	97	ØØØ11ØØØØ1	Ø61	15Ø	ØØ1ØØ1Ø11Ø	Ø96
45	ØØØØ1Ø11Ø1	Ø2D	98	ØØØ11ØØØ1Ø	Ø62	151	ØØ1ØØ1Ø111	Ø97
46	ØØØØ1Ø111Ø	Ø2E	99	ØØØ11ØØØ11	Ø63	152	ØØ1ØØ11ØØØ	Ø98
47	ØØØØ1Ø1111	Ø2F	1ØØ	ØØØ11ØØ1ØØ	Ø64	153	ØØ1ØØ11ØØ1	Ø99
48	ØØØØ11ØØØØ	Ø3Ø	1Ø1	ØØØ11ØØ1Ø1	Ø65	154	ØØ1ØØ11Ø1Ø	Ø9A
49	ØØØØ11ØØØ1	Ø31	1Ø2	ØØØ11ØØ11Ø	Ø66	155	ØØ1ØØ11Ø11	Ø9B
5Ø	ØØØØ11ØØ1Ø	Ø32	1Ø3	ØØØ11ØØ111	Ø67	156	ØØ1ØØ111ØØ	Ø9C
51	ØØØØ11ØØ11	Ø33	1Ø4	ØØØ11Ø1ØØØ	Ø68	157	ØØ1ØØ111Ø1	Ø9D
52	ØØØØ11Ø1ØØ	Ø34	1Ø5	ØØØ11Ø1ØØ1	Ø69	158	ØØ1ØØ1111Ø	Ø9E

DEC	BINARY	HEX	DEC	BINARY	HEX	DEC	BINARY	HEX
159	ØØ1ØØ11111	Ø9F	212	ØØ11Ø1Ø1ØØ	ØD4	265	Ø1ØØØØ1ØØ1	1Ø9
16Ø	ØØ1Ø1ØØØØØ	ØAØ	213	ØØ11Ø1Ø1Ø1	ØD5	266	Ø1ØØØØ1Ø1Ø	1ØA
161	ØØ1Ø1ØØØØ1	ØA1	214	ØØ11Ø1Ø11Ø	ØD6	267	Ø1ØØØØ1Ø11	1ØB
162	ØØ1Ø1ØØØ1Ø	ØA2	215	ØØ11Ø1Ø111	ØD7	268	Ø1ØØØØ11ØØ	1ØC
163	ØØ1Ø1ØØØ11	ØA3	216	ØØ11Ø11ØØØ	ØD8	269	Ø1ØØØØ11Ø1	1ØD
164	ØØ1Ø1ØØ1ØØ	ØA4	217	ØØ11Ø11ØØ1	ØD9	27Ø	Ø1ØØØØ111Ø	1ØE
165	ØØ1Ø1ØØ1Ø1	ØA5	218	ØØ11Ø11Ø1Ø	ØDA	271	Ø1ØØØØ1111	1ØF
166	ØØ1Ø1ØØ11Ø	ØA6	219	ØØ11Ø11Ø11	ØDB	272	Ø1ØØØ1ØØØØ	11Ø
167	ØØ1Ø1ØØ111	ØA7	22Ø	ØØ11Ø111ØØ	ØDC	273	Ø1ØØØ1ØØØ1	111
168	ØØ1Ø1Ø1ØØØ	ØA8	221	ØØ11Ø111Ø1	ØDD	274	Ø1ØØØ1ØØ1Ø	112
169	ØØ1Ø1Ø1ØØ1	ØA9	222	ØØ11Ø1111Ø	ØDE	275	Ø1ØØØ1ØØ11	113
17Ø	ØØ1Ø1Ø1Ø1Ø	ØAA	223	ØØ11Ø11111	ØDF	276	Ø1ØØØ1Ø1ØØ	114
171	ØØ1Ø1Ø1Ø11	ØAB	224	ØØ111ØØØØØ	ØEØ	277	Ø1ØØØ1Ø1Ø1	115
172	ØØ1Ø1Ø11ØØ	ØAC	225	ØØ111ØØØØ1	ØE1	278	Ø1ØØØ1Ø11Ø	116
173	ØØ1Ø1Ø11Ø1	ØAD	226	ØØ111ØØØ1Ø	ØE2	279	Ø1ØØØ1Ø111	117
174	ØØ1Ø1Ø111Ø	ØAE	227	ØØ111ØØØ11	ØE3	28Ø	Ø1ØØØ11ØØØ	118
175	ØØ1Ø1Ø1111	ØAF	228	ØØ111ØØ1ØØ	ØE4	281	Ø1ØØØ11ØØ1	119
176	ØØ1Ø11ØØØØ	ØBØ	229	ØØ111ØØ1Ø1	ØE5	282	Ø1ØØØ11Ø1Ø	11A
177	ØØ1Ø11ØØØ1	ØB1	23Ø	ØØ111ØØ11Ø	ØE6	283	Ø1ØØØ11Ø11	11B
178	ØØ1Ø11ØØ1Ø	ØB2	231	ØØ111ØØ111	ØE7	284	Ø1ØØØ111ØØ	11C
179	ØØ1Ø11ØØ11	ØB3	232	ØØ111Ø1ØØØ	ØE8	285	Ø1ØØØ111Ø1	11D
18Ø	ØØ1Ø11Ø1ØØ	ØB4	233	ØØ111Ø1ØØ1	ØE9	286	Ø1ØØØ1111Ø	11E
181	ØØ1Ø11Ø1Ø1	ØB5	234	ØØ111Ø1Ø1Ø	ØEA	287	Ø1ØØØ11111	11F
182	ØØ1Ø11Ø11Ø	ØB6	235	ØØ111Ø1Ø11	ØEB	288	Ø1ØØ1ØØØØØ	12Ø
183	ØØ1Ø11Ø111	ØB7	236	ØØ111Ø11ØØ	ØEC	289	Ø1ØØ1ØØØØ1	121
184	ØØ1Ø111ØØØ	ØB8	237	ØØ111Ø11Ø1	ØED	29Ø	Ø1ØØ1ØØØ1Ø	122
185	ØØ1Ø111ØØ1	ØB9	238	ØØ111Ø111Ø	ØEE	291	Ø1ØØ1ØØØ11	123
186	ØØ1Ø111Ø1Ø	ØBA	239	ØØ111Ø1111	ØEF	292	Ø1ØØ1ØØ1ØØ	124
187	ØØ1Ø111Ø11	ØBB	24Ø	ØØ1111ØØØØ	ØFØ	293	Ø1ØØ1ØØ1Ø1	125
188	ØØ1Ø1111ØØ	ØBC	241	ØØ1111ØØØ1	ØF1	294	Ø1ØØ1ØØ11Ø	126
189	ØØ1Ø1111Ø1	ØBD	242	ØØ1111ØØ1Ø	ØF2	295	Ø1ØØ1ØØ111	127
19Ø	ØØ1Ø11111Ø	ØBE	243	ØØ1111ØØ11	ØF3	296	Ø1ØØ1Ø1ØØØ	128
191	ØØ1Ø111111	ØBF	244	ØØ1111Ø1ØØ	ØF4	297	Ø1ØØ1Ø1ØØ1	129
192	ØØ11ØØØØØØ	ØCØ	245	ØØ1111Ø1Ø1	ØF5	298	Ø1ØØ1Ø1Ø1Ø	12A
193	ØØ11ØØØØØ1	ØC1	246	ØØ1111Ø11Ø	ØF6	299	Ø1ØØ1Ø1Ø11	12B
194	ØØ11ØØØØ1Ø	ØC2	247	ØØ1111Ø111	ØF7	3ØØ	Ø1ØØ1Ø11ØØ	12C
195	ØØ11ØØØØ11	ØC3	248	ØØ11111ØØØ	ØF8	3Ø1	Ø1ØØ1Ø11Ø1	12D
196	ØØ11ØØØ1ØØ	ØC4	249	ØØ11111ØØ1	ØF9	3Ø2	Ø1ØØ1Ø111Ø	12E
197	ØØ11ØØØ1Ø1	ØC5	25Ø	ØØ11111Ø1Ø	ØFA	3Ø3	Ø1ØØ1Ø1111	12F
198	ØØ11ØØØ11Ø	ØC6	251	ØØ11111Ø11	ØFB	3Ø4	Ø1ØØ11ØØØØ	13Ø
199	ØØ11ØØØ111	ØC7	252	ØØ111111ØØ	ØFC	3Ø5	Ø1ØØ11ØØØ1	131
2ØØ	ØØ11ØØ1ØØØ	ØC8	253	ØØ111111Ø1	ØFD	3Ø6	Ø1ØØ11ØØ1Ø	132
2Ø1	ØØ11ØØ1ØØ1	ØC9	254	ØØ1111111Ø	ØFE	3Ø7	Ø1ØØ11ØØ11	133
2Ø2	ØØ11ØØ1Ø1Ø	ØCA	255	ØØ11111111	ØFF	3Ø8	Ø1ØØ11Ø1ØØ	134
2Ø3	ØØ11ØØ1Ø11	ØCB	256	Ø1ØØØØØØØØ	1ØØ	3Ø9	Ø1ØØ11Ø1Ø1	135
2Ø4	ØØ11ØØ11ØØ	ØCC	257	Ø1ØØØØØØØ1	1Ø1	31Ø	Ø1ØØ11Ø11Ø	136
2Ø5	ØØ11ØØ11Ø1	ØCD	258	Ø1ØØØØØØ1Ø	1Ø2	311	Ø1ØØ11Ø111	137
2Ø6	ØØ11ØØ111Ø	ØCE	259	Ø1ØØØØØØ11	1Ø3	312	Ø1ØØ111ØØØ	138
2Ø7	ØØ11ØØ1111	ØCF	26Ø	Ø1ØØØØØ1ØØ	1Ø4	313	Ø1ØØ111ØØ1	139
2Ø8	ØØ11Ø1ØØØØ	ØDØ	261	Ø1ØØØØØ1Ø1	1Ø5	314	Ø1ØØ111Ø1Ø	13A
2Ø9	ØØ11Ø1ØØØ1	ØD1	262	Ø1ØØØØØ11Ø	1Ø6	315	Ø1ØØ111Ø11	13B
21Ø	ØØ11Ø1ØØ1Ø	ØD2	263	Ø1ØØØØØ111	1Ø7	316	Ø1ØØ1111ØØ	13C
211	ØØ11Ø1ØØ11	ØD3	264	Ø1ØØØØ1ØØØ	1Ø8	317	Ø1ØØ1111Ø1	13D

DEC	BINARY	HEX	DEC	BINARY	HEX	DEC	BINARY	HEX
318	Ø1ØØ11111Ø	13E	371	Ø1Ø111ØØ11	173	424	Ø11Ø1Ø1ØØØ	1A8
319	Ø1ØØ111111	13F	372	Ø1Ø111Ø1ØØ	174	425	Ø11Ø1Ø1ØØ1	1A9
32Ø	Ø1Ø1ØØØØØØ	14Ø	373	Ø1Ø111Ø1Ø1	175	426	Ø11Ø1Ø1Ø1Ø	1AA
321	Ø1Ø1ØØØØØ1	141	374	Ø1Ø111Ø11Ø	176	427	Ø11Ø1Ø1Ø11	1AB
322	Ø1Ø1ØØØØ1Ø	142	375	Ø1Ø111Ø111	177	428	Ø11Ø1Ø11ØØ	1AC
323	Ø1Ø1ØØØØ11	143	376	Ø1Ø1111ØØØ	178	429	Ø11Ø1Ø11Ø1	1AD
324	Ø1Ø1ØØØ1ØØ	144	377	Ø1Ø1111ØØ1	179	43Ø	Ø11Ø1Ø111Ø	1AE
325	Ø1Ø1ØØØ1Ø1	145	378	Ø1Ø1111Ø1Ø	17A	431	Ø11Ø1Ø1111	1AF
326	Ø1Ø1ØØØ11Ø	146	379	Ø1Ø1111Ø11	17B	432	Ø11Ø11ØØØØ	1BØ
327	Ø1Ø1ØØØ111	147	38Ø	Ø1Ø11111ØØ	17C	433	Ø11Ø11ØØØ1	1B1
328	Ø1Ø1ØØ1ØØØ	148	381	Ø1Ø11111Ø1	17D	434	Ø11Ø11ØØ1Ø	1B2
329	Ø1Ø1ØØ1ØØ1	149	382	Ø1Ø111111Ø	17E	435	Ø11Ø11ØØ11	1B3
33Ø	Ø1Ø1ØØ1Ø1Ø	14A	383	Ø1Ø1111111	17F	436	Ø11Ø11Ø1ØØ	1B4
331	Ø1Ø1ØØ1Ø11	14B	384	Ø11ØØØØØØØ	18Ø	437	Ø11Ø11Ø1Ø1	1B5
332	Ø1Ø1ØØ11ØØ	14C	385	Ø11ØØØØØØ1	181	438	Ø11Ø11Ø11Ø	1B6
333	Ø1Ø1ØØ11Ø1	14D	386	Ø11ØØØØØ1Ø	182	439	Ø11Ø11Ø111	1B7
334	Ø1Ø1ØØ111Ø	14E	387	Ø11ØØØØØ11	183	44Ø	Ø11Ø111ØØØ	1B8
335	Ø1Ø1ØØ1111	14F	388	Ø11ØØØØ1ØØ	184	441	Ø11Ø111ØØ1	1B9
336	Ø1Ø1Ø1ØØØØ	15Ø	389	Ø11ØØØØ1Ø1	185	442	Ø11Ø111Ø1Ø	1BA
337	Ø1Ø1Ø1ØØØ1	151	39Ø	Ø11ØØØØ11Ø	186	443	Ø11Ø111Ø11	1BB
338	Ø1Ø1Ø1ØØ1Ø	152	391	Ø11ØØØØ111	187	444	Ø11Ø1111ØØ	1BC
339	Ø1Ø1Ø1ØØ11	153	392	Ø11ØØØ1ØØØ	188	445	Ø11Ø1111Ø1	1BD
34Ø	Ø1Ø1Ø1Ø1ØØ	154	393	Ø11ØØØ1ØØ1	189	446	Ø11Ø11111Ø	1BE
341	Ø1Ø1Ø1Ø1Ø1	155	394	Ø11ØØØ1Ø1Ø	18A	447	Ø11Ø111111	1BF
342	Ø1Ø1Ø1Ø11Ø	156	395	Ø11ØØØ1Ø11	18B	448	Ø111ØØØØØØ	1CØ
343	Ø1Ø1Ø1Ø111	157	396	Ø11ØØØ11ØØ	18C	449	Ø111ØØØØØ1	1C1
344	Ø1Ø1Ø11ØØØ	158	397	Ø11ØØØ11Ø1	18D	45Ø	Ø111ØØØØ1Ø	1C2
345	Ø1Ø1Ø11ØØ1	159	398	Ø11ØØØ111Ø	18E	451	Ø111ØØØØ11	1C3
346	Ø1Ø1Ø11Ø1Ø	15A	399	Ø11ØØØ1111	18F	452	Ø111ØØØ1ØØ	1C4
347	Ø1Ø1Ø11Ø11	15B	4ØØ	Ø11ØØ1ØØØØ	19Ø	453	Ø111ØØØ1Ø1	1C5
348	Ø1Ø1Ø111ØØ	15C	4Ø1	Ø11ØØ1ØØØ1	191	454	Ø111ØØØ11Ø	1C6
349	Ø1Ø1Ø111Ø1	15D	4Ø2	Ø11ØØ1ØØ1Ø	192	455	Ø111ØØØ111	1C7
35Ø	Ø1Ø1Ø1111Ø	15E	4Ø3	Ø11ØØ1ØØ11	193	456	Ø111ØØ1ØØØ	1C8
351	Ø1Ø1Ø11111	15F	4Ø4	Ø11ØØ1Ø1ØØ	194	457	Ø111ØØ1ØØ1	1C9
352	Ø1Ø11ØØØØØ	16Ø	4Ø5	Ø11ØØ1Ø1Ø1	195	458	Ø111ØØ1Ø1Ø	1CA
353	Ø1Ø11ØØØØ1	161	4Ø6	Ø11ØØ1Ø11Ø	196	459	Ø111ØØ1Ø11	1CB
354	Ø1Ø11ØØØ1Ø	162	4Ø7	Ø11ØØ1Ø111	197	46Ø	Ø111ØØ11ØØ	1CC
355	Ø1Ø11ØØØ11	163	4Ø8	Ø11ØØ11ØØØ	198	461	Ø111ØØ11Ø1	1CD
356	Ø1Ø11ØØ1ØØ	164	4Ø9	Ø11ØØ11ØØ1	199	462	Ø111ØØ111Ø	1CE
357	Ø1Ø11ØØ1Ø1	165	41Ø	Ø11ØØ11Ø1Ø	19A	463	Ø111ØØ1111	1CF
358	Ø1Ø11ØØ11Ø	166	411	Ø11ØØ11Ø11	19B	464	Ø111Ø1ØØØØ	1DØ
359	Ø1Ø11ØØ111	167	412	Ø11ØØ111ØØ	19C	465	Ø111Ø1ØØØ1	1D1
36Ø	Ø1Ø11Ø1ØØØ	168	413	Ø11ØØ111Ø1	19D	466	Ø111Ø1ØØ1Ø	1D2
361	Ø1Ø11Ø1ØØ1	169	414	Ø11ØØ1111Ø	19E	467	Ø111Ø1ØØ11	1D3
362	Ø1Ø11Ø1Ø1Ø	16A	415	Ø11ØØ11111	19F	468	Ø111Ø1Ø1ØØ	1D4
363	Ø1Ø11Ø1Ø11	16B	416	Ø11Ø1ØØØØØ	1AØ	469	Ø111Ø1Ø1Ø1	1D5
364	Ø1Ø11Ø11ØØ	16C	417	Ø11Ø1ØØØØ1	1A1	47Ø	Ø111Ø1Ø11Ø	1D6
365	Ø1Ø11Ø11Ø1	16D	418	Ø11Ø1ØØØ1Ø	1A2	471	Ø111Ø1Ø111	1D7
366	Ø1Ø11Ø111Ø	16E	419	Ø11Ø1ØØØ11	1A3	472	Ø111Ø11ØØØ	1D8
367	Ø1Ø11Ø1111	16F	42Ø	Ø11Ø1ØØ1ØØ	1A4	473	Ø111Ø11ØØ1	1D9
368	Ø1Ø111ØØØØ	17Ø	421	Ø11Ø1ØØ1Ø1	1A5	474	Ø111Ø11Ø1Ø	1DA
369	Ø1Ø111ØØØ1	171	422	Ø11Ø1ØØ11Ø	1A6	475	Ø111Ø11Ø11	1DB
37Ø	Ø1Ø111ØØ1Ø	172	423	Ø11Ø1ØØ111	1A7	476	Ø111Ø111ØØ	1DC

DEC	BINARY	HEX	DEC	BINARY	HEX	DEC	BINARY	HEX
477	Ø111Ø111Ø1	1DD	53Ø	1ØØØØ1ØØ1Ø	212	583	1ØØ1ØØØ111	247
478	Ø111Ø1111Ø	1DE	531	1ØØØØ1ØØ11	213	584	1ØØ1ØØ1ØØØ	248
479	Ø111Ø11111	1DF	532	1ØØØØ1Ø1ØØ	214	585	1ØØ1ØØ1ØØ1	249
48Ø	Ø1111ØØØØØ	1EØ	533	1ØØØØ1Ø1Ø1	215	586	1ØØ1ØØ1Ø1Ø	24A
481	Ø1111ØØØØ1	1E1	534	1ØØØØ1Ø11Ø	216	587	1ØØ1ØØ1Ø11	24B
482	Ø1111ØØØ1Ø	1E2	535	1ØØØØ1Ø111	217	588	1ØØ1ØØ11ØØ	24C
483	Ø1111ØØØ11	1E3	536	1ØØØØ11ØØØ	218	589	1ØØ1ØØ11Ø1	24D
484	Ø1111ØØ1ØØ	1E4	537	1ØØØØ11ØØ1	219	59Ø	1ØØ1ØØ111Ø	24E
485	Ø1111ØØ1Ø1	1E5	538	1ØØØØ11Ø1Ø	21A	591	1ØØ1ØØ1111	24F
486	Ø1111ØØ11Ø	1E6	539	1ØØØØ11Ø11	21B	592	1ØØ1Ø1ØØØØ	25Ø
487	Ø1111ØØ111	1E7	54Ø	1ØØØØ111ØØ	21C	593	1ØØ1Ø1ØØØ1	251
488	Ø1111Ø1ØØØ	1E8	541	1ØØØØ111Ø1	21D	594	1ØØ1Ø1ØØ1Ø	252
489	Ø1111Ø1ØØ1	1E9	542	1ØØØØ1111Ø	21E	595	1ØØ1Ø1ØØ11	253
49Ø	Ø1111Ø1Ø1Ø	1EA	543	1ØØØØ11111	21F	596	1ØØ1Ø1Ø1ØØ	254
491	Ø1111Ø1Ø11	1EB	544	1ØØØ1ØØØØØ	22Ø	597	1ØØ1Ø1Ø1Ø1	255
492	Ø1111Ø11ØØ	1EC	545	1ØØØ1ØØØØ1	221	598	1ØØ1Ø1Ø11Ø	256
493	Ø1111Ø11Ø1	1ED	546	1ØØØ1ØØØ1Ø	222	599	1ØØ1Ø1Ø111	257
494	Ø1111Ø111Ø	1EE	547	1ØØØ1ØØØ11	223	6ØØ	1ØØ1Ø11ØØØ	258
495	Ø1111Ø1111	1EF	548	1ØØØ1ØØ1ØØ	224	6Ø1	1ØØ1Ø11ØØ1	259
496	Ø11111ØØØØ	1FØ	549	1ØØØ1ØØ1Ø1	225	6Ø2	1ØØ1Ø11Ø1Ø	25A
497	Ø11111ØØØ1	1F1	55Ø	1ØØØ1ØØ11Ø	226	6Ø3	1ØØ1Ø11Ø11	25B
498	Ø11111ØØ1Ø	1F2	551	1ØØØ1ØØ111	227	6Ø4	1ØØ1Ø111ØØ	25C
499	Ø11111ØØ11	1F3	552	1ØØØ1Ø1ØØØ	228	6Ø5	1ØØ1Ø111Ø1	25D
5ØØ	Ø11111Ø1ØØ	1F4	553	1ØØØ1Ø1ØØ1	229	6Ø6	1ØØ1Ø1111Ø	25E
5Ø1	Ø11111Ø1Ø1	1F5	554	1ØØØ1Ø1Ø1Ø	22A	6Ø7	1ØØ1Ø11111	25F
5Ø2	Ø11111Ø11Ø	1F6	555	1ØØØ1Ø1Ø11	22B	6Ø8	1ØØ11ØØØØØ	26Ø
5Ø3	Ø11111Ø111	1F7	556	1ØØØ1Ø11ØØ	22C	6Ø9	1ØØ11ØØØØ1	261
5Ø4	Ø111111ØØØ	1F8	557	1ØØØ1Ø11Ø1	22D	61Ø	1ØØ11ØØØ1Ø	262
5Ø5	Ø111111ØØ1	1F9	558	1ØØØ1Ø111Ø	22E	611	1ØØ11ØØØ11	263
5Ø6	Ø111111Ø1Ø	1FA	559	1ØØØ1Ø1111	22F	612	1ØØ11ØØ1ØØ	264
5Ø7	Ø111111Ø11	1FB	56Ø	1ØØØ11ØØØØ	23Ø	613	1ØØ11ØØ1Ø1	265
5Ø8	Ø1111111ØØ	1FC	561	1ØØØ11ØØØ1	231	614	1ØØ11ØØ11Ø	266
5Ø9	Ø1111111Ø1	1FD	562	1ØØØ11ØØ1Ø	232	615	1ØØ11ØØ111	267
51Ø	Ø11111111Ø	1FE	563	1ØØØ11ØØ11	233	616	1ØØ11Ø1ØØØ	268
511	Ø111111111	1FF	564	1ØØØ11Ø1ØØ	234	617	1ØØ11Ø1ØØ1	269
512	1ØØØØØØØØØ	2ØØ	565	1ØØØ11Ø1Ø1	235	618	1ØØ11Ø1Ø1Ø	26A
513	1ØØØØØØØØ1	2Ø1	566	1ØØØ11Ø11Ø	236	619	1ØØ11Ø1Ø11	26B
514	1ØØØØØØØ1Ø	2Ø2	567	1ØØØ11Ø111	237	62Ø	1ØØ11Ø11ØØ	26C
515	1ØØØØØØØ11	2Ø3	568	1ØØØ111ØØØ	238	621	1ØØ11Ø11Ø1	26D
516	1ØØØØØØ1ØØ	2Ø4	569	1ØØØ111ØØ1	239	622	1ØØ11Ø111Ø	26E
517	1ØØØØØØ1Ø1	2Ø5	57Ø	1ØØØ111Ø1Ø	23A	623	1ØØ11Ø1111	26F
518	1ØØØØØØ11Ø	2Ø6	571	1ØØØ111Ø11	23B	624	1ØØ111ØØØØ	27Ø
519	1ØØØØØØ111	2Ø7	572	1ØØØ1111ØØ	23C	625	1ØØ111ØØØ1	271
52Ø	1ØØØØØ1ØØØ	2Ø8	573	1ØØØ1111Ø1	23D	626	1ØØ111ØØ1Ø	272
521	1ØØØØØ1ØØ1	2Ø9	574	1ØØØ11111Ø	23E	627	1ØØ111ØØ11	273
522	1ØØØØØ1Ø1Ø	2ØA	575	1ØØØ111111	23F	628	1ØØ111Ø1ØØ	274
523	1ØØØØØ1Ø11	2ØB	576	1ØØ1ØØØØØØ	24Ø	629	1ØØ111Ø1Ø1	275
524	1ØØØØØ11ØØ	2ØC	577	1ØØ1ØØØØØ1	241	63Ø	1ØØ111Ø11Ø	276
525	1ØØØØØ11Ø1	2ØD	578	1ØØ1ØØØØ1Ø	242	631	1ØØ111Ø111	277
526	1ØØØØØ111Ø	2ØE	579	1ØØ1ØØØØ11	243	632	1ØØ1111ØØØ	278
527	1ØØØØØ1111	2ØF	58Ø	1ØØ1ØØØ1ØØ	244	633	1ØØ1111ØØ1	279
528	1ØØØØ1ØØØØ	21Ø	581	1ØØ1ØØØ1Ø1	245	634	1ØØ1111Ø1Ø	27A
529	1ØØØØ1ØØØ1	211	582	1ØØ1ØØØ11Ø	246	635	1ØØ1111Ø11	27B

DEC	BINARY	HEX	DEC	BINARY	HEX	DEC	BINARY	HEX
636	1001111100	27C	689	1010110001	2B1	742	1011100110	2E6
637	1001111101	27D	690	1010110010	2B2	743	1011100111	2E7
638	1001111110	27E	691	1010110011	2B3	744	1011101000	2E8
639	1001111111	27F	692	1010110100	2B4	745	1011101001	2E9
640	1010000000	280	693	1010110101	2B5	746	1011101010	2EA
641	1010000001	281	694	1010110110	2B6	747	1011101011	2EB
642	1010000010	282	695	1010110111	2B7	748	1011101100	2EC
643	1010000011	283	696	1010111000	2B8	749	1011101101	2ED
644	1010000100	284	697	1010111001	2B9	750	1011101110	2EE
645	1010000101	285	698	1010111010	2BA	751	1011101111	2EF
646	1010000110	286	699	1010111011	2BB	752	1011110000	2F0
647	1010000111	287	700	1010111100	2BC	753	1011110001	2F1
648	1010001000	288	701	1010111101	2BD	754	1011110010	2F2
649	1010001001	289	702	1010111110	2BE	755	1011110011	2F3
650	1010001010	28A	703	1010111111	2BF	756	1011110100	2F4
651	1010001011	28B	704	1011000000	2C0	757	1011110101	2F5
652	1010001100	28C	705	1011000001	2C1	758	1011110110	2F6
653	1010001101	28D	706	1011000010	2C2	759	1011110111	2F7
654	1010001110	28E	707	1011000011	2C3	760	1011111000	2F8
655	1010001111	28F	708	1011000100	2C4	761	1011111001	2F9
656	1010010000	290	709	1011000101	2C5	762	1011111010	2FA
657	1010010001	291	710	1011000110	2C6	763	1011111011	2FB
658	1010010010	292	711	1011000111	2C7	764	1011111100	2FC
659	1010010011	293	712	1011001000	2C8	765	1011111101	2FD
660	1010010100	294	713	1011001001	2C9	766	1011111110	2FE
661	1010010101	295	714	1011001010	2CA	767	1011111111	2FF
662	1010010110	296	715	1011001011	2CB	768	1100000000	300
663	1010010111	297	716	1011001100	2CC	769	1100000001	301
664	1010011000	298	717	1011001101	2CD	770	1100000010	302
665	1010011001	299	718	1011001110	2CE	771	1100000011	303
666	1010011010	29A	719	1011001111	2CF	772	1100000100	304
667	1010011011	29B	720	1011010000	2D0	773	1100000101	305
668	1010011100	29C	721	1011010001	2D1	774	1100000110	306
669	1010011101	29D	722	1011010010	2D2	775	1100000111	307
670	1010011110	29E	723	1011010011	2D3	776	1100001000	308
671	1010011111	29F	724	1011010100	2D4	777	1100001001	309
672	1010100000	2A0	725	1011010101	2D5	778	1100001010	30A
673	1010100001	2A1	726	1011010110	2D6	779	1100001011	30B
674	1010100010	2A2	727	1011010111	2D7	780	1100001100	30C
675	1010100011	2A3	728	1011011000	2D8	781	1100001101	30D
676	1010100100	2A4	729	1011011001	2D9	782	1100001110	30E
677	1010100101	2A5	730	1011011010	2DA	783	1100001111	30F
678	1010100110	2A6	731	1011011011	2DB	784	1100010000	310
679	1010100111	2A7	732	1011011100	2DC	785	1100010001	311
680	1010101000	2A8	733	1011011101	2DD	786	1100010010	312
681	1010101001	2A9	734	1011011110	2DE	787	1100010011	313
682	1010101010	2AA	735	1011011111	2DF	788	1100010100	314
683	1010101011	2AB	736	1011100000	2E0	789	1100010101	315
684	1010101100	2AC	737	1011100001	2E1	790	1100010110	316
685	1010101101	2AD	738	1011100010	2E2	791	1100010111	317
686	1010101110	2AE	739	1011100011	2E3	792	1100011000	318
687	1010101111	2AF	740	1011100100	2E4	793	1100011001	319
688	1010110000	2B0	741	1011100101	2E5	794	1100011010	31A

DEC	BINARY	HEX	DEC	BINARY	HEX	DEC	BINARY	HEX
795	11ØØØ11Ø11	31B	848	11Ø1Ø1ØØØØ	35Ø	9Ø1	111ØØØØ1Ø1	385
796	11ØØØ111ØØ	31C	849	11Ø1Ø1ØØØ1	351	9Ø2	111ØØØØ11Ø	386
797	11ØØØ111Ø1	31D	85Ø	11Ø1Ø1ØØ1Ø	352	9Ø3	111ØØØØ111	387
798	11ØØØ1111Ø	31E	851	11Ø1Ø1ØØ11	353	9Ø4	111ØØØ1ØØØ	388
799	11ØØØ11111	31F	852	11Ø1Ø1Ø1ØØ	354	9Ø5	111ØØØ1ØØ1	389
8ØØ	11ØØ1ØØØØØ	32Ø	853	11Ø1Ø1Ø1Ø1	355	9Ø6	111ØØØ1Ø1Ø	38A
8Ø1	11ØØ1ØØØØ1	321	854	11Ø1Ø1Ø11Ø	356	9Ø7	111ØØØ1Ø11	38B
8Ø2	11ØØ1ØØØ1Ø	322	855	11Ø1Ø1Ø111	357	9Ø8	111ØØØ11ØØ	38C
8Ø3	11ØØ1ØØØ11	323	856	11Ø1Ø11ØØØ	358	9Ø9	111ØØØ11Ø1	38D
8Ø4	11ØØ1ØØ1ØØ	324	857	11Ø1Ø11ØØ1	359	91Ø	111ØØØ111Ø	38E
8Ø5	11ØØ1ØØ1Ø1	325	858	11Ø1Ø11Ø1Ø	35A	911	111ØØØ1111	38F
8Ø6	11ØØ1ØØ11Ø	326	859	11Ø1Ø11Ø11	35B	912	111ØØ1ØØØØ	39Ø
8Ø7	11ØØ1ØØ111	327	86Ø	11Ø1Ø111ØØ	35C	913	111ØØ1ØØØ1	391
8Ø8	11ØØ1Ø1ØØØ	328	861	11Ø1Ø111Ø1	35D	914	111ØØ1ØØ1Ø	392
8Ø9	11ØØ1Ø1ØØ1	329	862	11Ø1Ø1111Ø	35E	915	111ØØ1ØØ11	393
81Ø	11ØØ1Ø1Ø1Ø	32A	863	11Ø1Ø11111	35F	916	111ØØ1Ø1ØØ	394
811	11ØØ1Ø1Ø11	32B	864	11Ø11ØØØØØ	36Ø	917	111ØØ1Ø1Ø1	395
812	11ØØ1Ø11ØØ	32C	865	11Ø11ØØØØ1	361	918	111ØØ1Ø11Ø	396
813	11ØØ1Ø11Ø1	32D	866	11Ø11ØØØ1Ø	362	919	111ØØ1Ø111	397
814	11ØØ1Ø111Ø	32E	867	11Ø11ØØØ11	363	92Ø	111ØØ11ØØØ	398
815	11ØØ1Ø1111	32F	868	11Ø11ØØ1ØØ	364	921	111ØØ11ØØ1	399
816	11ØØ11ØØØØ	33Ø	869	11Ø11ØØ1Ø1	365	922	111ØØ11Ø1Ø	39A
817	11ØØ11ØØØ1	331	87Ø	11Ø11ØØ11Ø	366	923	111ØØ11Ø11	39B
818	11ØØ11ØØ1Ø	332	871	11Ø11ØØ111	367	924	111ØØ111ØØ	39C
819	11ØØ11ØØ11	333	872	11Ø11Ø1ØØØ	368	925	111ØØ111Ø1	39D
82Ø	11ØØ11Ø1ØØ	334	873	11Ø11Ø1ØØ1	369	926	111ØØ1111Ø	39E
821	11ØØ11Ø1Ø1	335	874	11Ø11Ø1Ø1Ø	36A	927	111ØØ11111	39F
822	11ØØ11Ø11Ø	336	875	11Ø11Ø1Ø11	36B	928	111Ø1ØØØØØ	3AØ
823	11ØØ11Ø111	337	876	11Ø11Ø11ØØ	36C	929	111Ø1ØØØØ1	3A1
824	11ØØ111ØØØ	338	877	11Ø11Ø11Ø1	36D	93Ø	111Ø1ØØØ1Ø	3A2
825	11ØØ111ØØ1	339	878	11Ø11Ø111Ø	36E	931	111Ø1ØØØ11	3A3
826	11ØØ111Ø1Ø	33A	879	11Ø11Ø1111	36F	932	111Ø1ØØ1ØØ	3A4
827	11ØØ111Ø11	33B	88Ø	11Ø111ØØØØ	37Ø	933	111Ø1ØØ1Ø1	3A5
828	11ØØ1111ØØ	33C	881	11Ø111ØØØ1	371	934	111Ø1ØØ11Ø	3A6
829	11ØØ1111Ø1	33D	882	11Ø111ØØ1Ø	372	935	111Ø1ØØ111	3A7
83Ø	11ØØ11111Ø	33E	883	11Ø111ØØ11	373	936	111Ø1Ø1ØØØ	3A8
831	11ØØ111111	33F	884	11Ø111Ø1ØØ	374	937	111Ø1Ø1ØØ1	3A9
832	11Ø1ØØØØØØ	34Ø	885	11Ø111Ø1Ø1	375	938	111Ø1Ø1Ø1Ø	3AA
833	11Ø1ØØØØØ1	341	886	11Ø111Ø11Ø	376	939	111Ø1Ø1Ø11	3AB
834	11Ø1ØØØØ1Ø	342	887	11Ø111Ø111	377	94Ø	111Ø1Ø11ØØ	3AC
835	11Ø1ØØØØ11	343	888	11Ø1111ØØØ	378	941	111Ø1Ø11Ø1	3AD
836	11Ø1ØØØ1ØØ	344	889	11Ø1111ØØ1	379	942	111Ø1Ø111Ø	3AE
837	11Ø1ØØØ1Ø1	345	89Ø	11Ø1111Ø1Ø	37A	943	111Ø1Ø1111	3AF
838	11Ø1ØØØ11Ø	346	891	11Ø1111Ø11	37B	944	111Ø11ØØØØ	3BØ
839	11Ø1ØØØ111	347	892	11Ø11111ØØ	37C	945	111Ø11ØØØ1	3B1
84Ø	11Ø1ØØ1ØØØ	348	893	11Ø11111Ø1	37D	946	111Ø11ØØ1Ø	3B2
841	11Ø1ØØ1ØØ1	349	894	11Ø111111Ø	37E	947	111Ø11ØØ11	3B3
842	11Ø1ØØ1Ø1Ø	34A	895	11Ø1111111	37F	948	111Ø11Ø1ØØ	3B4
843	11Ø1ØØ1Ø11	34B	896	111ØØØØØØØ	38Ø	949	111Ø11Ø1Ø1	3B5
844	11Ø1ØØ11ØØ	34C	897	111ØØØØØØ1	381	95Ø	111Ø11Ø11Ø	3B6
845	11Ø1ØØ11Ø1	34D	898	111ØØØØØ1Ø	382	951	111Ø11Ø111	3B7
846	11Ø1ØØ111Ø	34E	899	111ØØØØØ11	383	952	111Ø111ØØØ	3B8
847	11Ø1ØØ1111	34F	9ØØ	111ØØØØ1ØØ	384	953	111Ø111ØØ1	3B9

DEC	BINARY	HEX	DEC	BINARY	HEX	DEC	BINARY	HEX
954	111Ø111Ø1Ø	3BA	977	1111Ø1ØØØ1	3D1	1ØØØ	11111Ø1ØØØ	3E8
955	111Ø111Ø11	3BB	978	1111Ø1ØØ1Ø	3D2	1ØØ1	11111Ø1ØØ1	3E9
956	111Ø1111ØØ	3BC	979	1111Ø1ØØ11	3D3	1ØØ2	11111Ø1Ø1Ø	3EA
957	111Ø1111Ø1	3BD	98Ø	1111Ø1Ø1ØØ	3D4	1ØØ3	11111Ø1Ø11	3EB
958	111Ø11111Ø	3BE	981	1111Ø1Ø1Ø1	3D5	1ØØ4	11111Ø11ØØ	3EC
959	111Ø111111	3BF	982	1111Ø1Ø11Ø	3D6	1ØØ5	11111Ø11Ø1	3ED
96Ø	1111ØØØØØØ	3CØ	983	1111Ø1Ø111	3D7	1ØØ6	11111Ø111Ø	3EE
961	1111ØØØØØ1	3C1	984	1111Ø11ØØØ	3D8	1ØØ7	11111Ø1111	3EF
962	1111ØØØØ1Ø	3C2	985	1111Ø11ØØ1	3D9	1ØØ8	111111ØØØØ	3FØ
963	1111ØØØØ11	3C3	986	1111Ø11Ø1Ø	3DA	1ØØ9	111111ØØØ1	3F1
964	1111ØØØ1ØØ	3C4	987	1111Ø11Ø11	3DB	1Ø1Ø	111111ØØ1Ø	3F2
965	1111ØØØ1Ø1	3C5	988	1111Ø111ØØ	3DC	1Ø11	111111ØØ11	3F3
966	1111ØØØ11Ø	3C6	989	1111Ø111Ø1	3DD	1Ø12	111111Ø1ØØ	3F4
967	1111ØØØ111	3C7	99Ø	1111Ø1111Ø	3DE	1Ø13	111111Ø1Ø1	3F5
968	1111ØØ1ØØØ	3C8	991	1111Ø11111	3DF	1Ø14	111111Ø11Ø	3F6
969	1111ØØ1ØØ1	3C9	992	11111ØØØØØ	3EØ	1Ø15	111111Ø111	3F7
97Ø	1111ØØ1Ø1Ø	3CA	993	11111ØØØØ1	3E1	1Ø16	1111111ØØØ	3F8
971	1111ØØ1Ø11	3CB	994	11111ØØØ1Ø	3E2	1Ø17	1111111ØØ1	3F9
972	1111ØØ11ØØ	3CC	995	11111ØØØ11	3E3	1Ø18	1111111Ø1Ø	3FA
973	1111ØØ11Ø1	3CD	996	11111ØØ1ØØ	3E4	1Ø19	1111111Ø11	3FB
974	1111ØØ111Ø	3CE	997	11111ØØ1Ø1	3E5	1Ø2Ø	11111111ØØ	3FC
975.	1111ØØ1111	3CF	998	11111ØØ11Ø	3E6	1Ø21	11111111Ø1	3FD
976	1111Ø1ØØØØ	3DØ	999	11111ØØ111	3E7	1Ø22	111111111Ø	3FE
						1Ø23	1111111111	3FF

CFDH
Converting from Decimal to Hexadecimal

If you have a decimal number you want to convert to a hexadecimal value, do the following:

1. If you are a Model II or Color Computer Extended BASIC or Disk user, use the HEX$ command to convert. PRINT HEX$(1000), for example, displays the hexadecimal equivalent of 1000.

2. If the decimal number is 0 to 1023, use Table CFDB-1.

3. If the number is greater than 1023 do this:

   A. Divide the number by 16. Save the remainder as R1.

   B. Divide the result of step A by 2 again. Save the remainder as R2.

   C. Repeat step B until the amount remaining is 0.

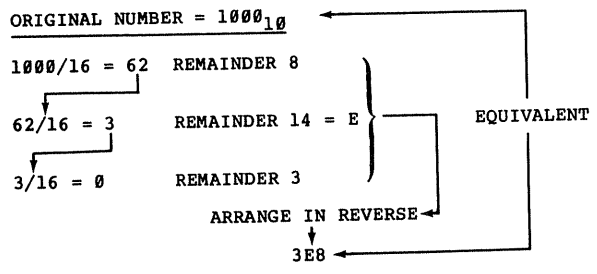
   D. Arrange the remainders in reverse order.

   E. Convert any remainder over 9 to the hexadecimal digits A through F (10 is A, 11 is B, 12 is C, 13 is D, 14 is E and 15 is F).

   F. The result is the equivalent binary number.

As an example: Convert 100 decimal to hexadecimal. See Figure CFDH-1 for the process.

Figure CFDH-1 -- *Converting from Decimal to Hexadecimal*



CFFT
/CMD File Format, TRSDOS, Models I/II/III

Figure CFFT-1 shows the /CMD file format for TRSDOS. You can see this format by using the F function in Model III Disk DEBUG (see DT1U) or by LISTing the file in hexadecimal.

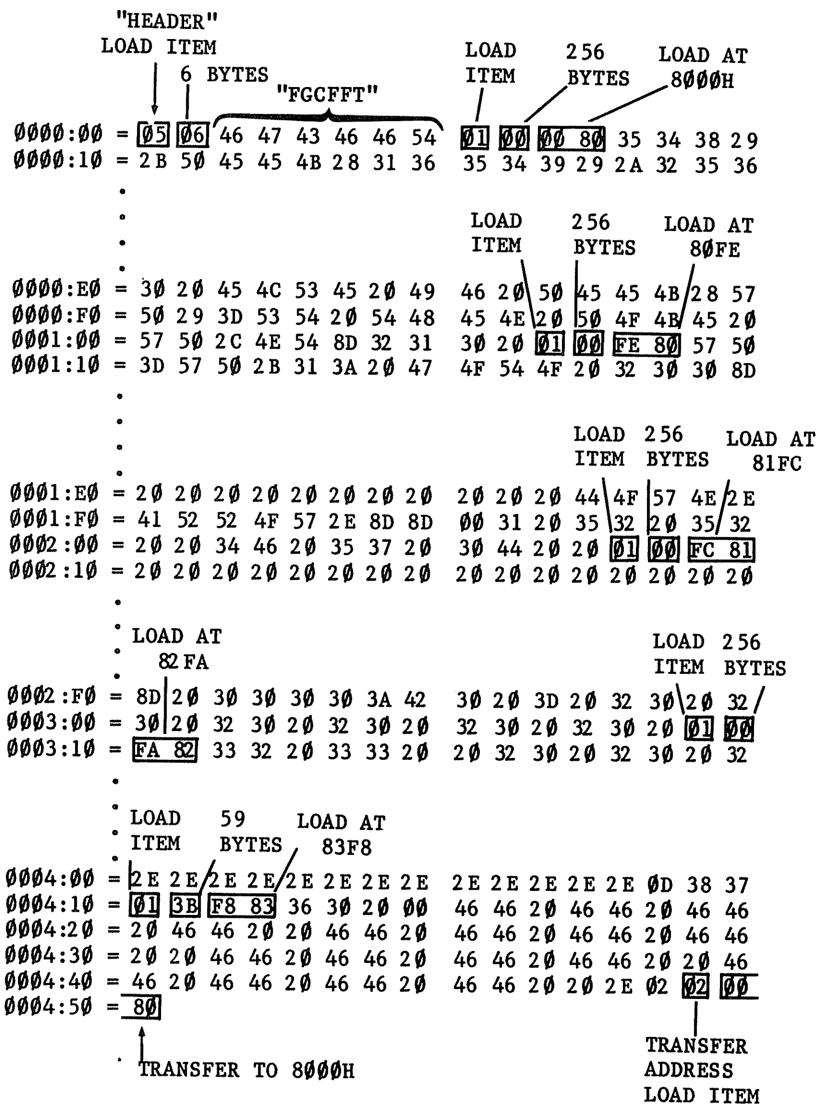
The first portion is present on some /CMD files, such as the /CMD files created by the TRSDOS DUMP command but not on other /CMD files, such as the /CMD files created by the Series I Editor/Assembler. It is an 18-byte "header" of hex 05 (5 decimal), followed by a byte containing a byte count of hex 10 (16 decimal), followed by descriptive text and creation date.

The following portions are "01" load items. These are a hex **01** (1 decimal) byte, followed by a byte containing a byte count, followed by two bytes containing a "load address", followed by data. The byte count defines how many bytes of data will be present. A value of **00** is 256 bytes of data, while values of **01** through **FF** represent 1 through 255 bytes of data (see CFHD for hexadecimal to decimal conversions). The "load address" is in standard Z-80 address format, least significant byte followed by most significant byte (see Z8AF). The first byte of data will be loaded at this address in RAM.

The "01" load items continue until the end of the data. After the last "01" portion, there is a "02" byte, followed by two bytes of a transfer address, in Z-80 address format. The "02" transfer address marks the end of the /CMD file.

A typical file might have "01" load items with a byte count of **00**, representing 256 bytes of data. Each "01" load item would be 258 bytes long, 256 bytes of data, plus the two bytes containing the "01" and length. Accessing the file by the DEBUG F command (see DT1U) or by a LIST would mean that the "01" load items would move slightly on the display. You can actually find the "01" load items fairly easily by using the F command. LDOS users: See DT1U, Extended Debug commands, N command.

Figure CFFT-1 – */CMD File Format, Models I/III*



```
ØØØØ:ØØ = Ø5 Ø6 46 47 43 46 46 54  Ø1 ØØ ØØ 8Ø 35 34 38 29
ØØØØ:1Ø = 2B 5Ø 45 45 4B 28 31 36  35 34 39 29 2A 32 35 36
        •
        •
        •
        •
ØØØØ:EØ = 3Ø 2Ø 45 4C 53 45 2Ø 49  46 2Ø 5Ø 45 45 4B 28 57
ØØØØ:FØ = 5Ø 29 3D 53 54 2Ø 54 48  45 4E 2Ø 5Ø 4F 4B 45 2Ø
ØØØ1:ØØ = 57 5Ø 2C 4E 54 8D 32 31  3Ø 2Ø Ø1 ØØ FE 8Ø 57 5Ø
ØØØ1:1Ø = 3D 57 5Ø 2B 31 3A 2Ø 47  4F 54 4F 2Ø 32 3Ø 3Ø 8D
        •
        •
        •
        •
ØØØ1:EØ = 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø  2Ø 2Ø 2Ø 44 4F 57 4E 2E
ØØØ1:FØ = 41 52 52 4F 57 2E 8D 8D  ØØ 31 2Ø 35 32 2Ø 35 32
ØØØ2:ØØ = 2Ø 2Ø 34 46 2Ø 35 37 2Ø  3Ø 44 2Ø 2Ø Ø1 ØØ FC 81
ØØØ2:1Ø = 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø  2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø
        •
        •
        •
        •
ØØØ2:FØ = 8D 2Ø 3Ø 3Ø 3Ø 3Ø 3A 42  3Ø 2Ø 3D 2Ø 32 3Ø 2Ø 32
ØØØ3:ØØ = 3Ø 2Ø 32 3Ø 2Ø 32 3Ø 2Ø  32 3Ø 2Ø 32 3Ø 2Ø Ø1 ØØ
ØØØ3:1Ø = FA 82 33 32 2Ø 33 33 2Ø  2Ø 32 3Ø 2Ø 32 3Ø 2Ø 32
        •
        •
        •
        •
ØØØ4:ØØ = 2E 2E 2E 2E 2E 2E 2E 2E  2E 2E 2E 2E 2E ØD 38 37
ØØØ4:1Ø = Ø1 3B F8 83 36 3Ø 2Ø ØØ  46 46 2Ø 46 46 2Ø 46 46
ØØØ4:2Ø = 2Ø 46 46 2Ø 2Ø 46 46 2Ø  46 46 2Ø 46 46 2Ø 46 46
ØØØ4:3Ø = 2Ø 2Ø 46 46 2Ø 46 46 2Ø  46 46 2Ø 46 46 2Ø 2Ø 46
ØØØ4:4Ø = 46 2Ø 46 46 2Ø 46 46 2Ø  46 46 2Ø 2Ø 2E Ø2 Ø2 ØØ
ØØØ4:5Ø = 8Ø
```

CFHD
Converting from Hexadecimal to Decimal

If you have a hexadecimal number that you would like to convert to decimal, do the following:

1. If the hexadecimal number is **0** through **3FF** use Table CFDB-1.

2. If the number is greater than **3FF** do the following:

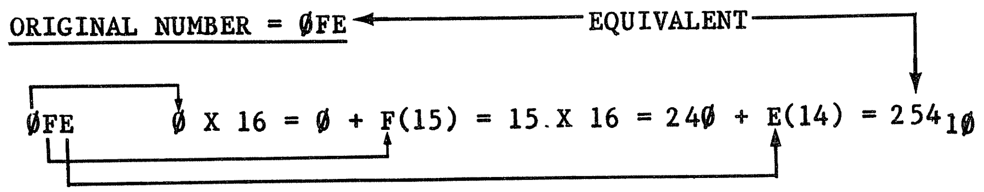
   A. Starting with the leftmost hex digit, multiply by 16. Add the *next* hex digit. Use 10, 11, 12, 13, 14, or 15 for hex digits of **A**, **B**, **C**, **D**, **E** and **F**.

   B. Multiply this result by 16 and add to the next hex digit.

   C. Repeat step B until the rightmost hex digit has been added. The result is the equivalent decimal number.

As an example: Convert **0FE** to decimal. See Figure CFHD-1.

Figure CFHD-1 – *Converting from Hexadecimal to Decimal*



CFOW
Cursor, Finding Out Where It Is In BASIC

All systems except Model I/III Level I and Color Computer Color BASIC:

Suppose you're doing a general input operation in BASIC and want to find out the current cursor position. The POS function will tell you where that blinkin' light is on the row:

```
1ØØ A=POS(Ø)
```

stores the current cursor position along the row in variable A. The "(0)" is a "dummy" argument that really doesn't do anything.

The position will correspond to the character position along the row and will be 0 through 63 for the Model I/III, 0 through 79 for the Model II, or 0 through 31 for the Color Computer.

POS is handy for columnating data (see CHTP) or for word processing applications.

Model II users only: ROW does the same for the cursor row position.

```
1ØØ B=ROW(Ø)
```

will store the current cursor row of 0 (top) through 23 (bottom) in variable B.

CFSD
Copying a File To Same Diskette, Model I/II/III TRSDOS, Color Computer

Use the COPY command to copy the file as follows:

```
COPY FILE1:Ø TO FILE2:Ø
```

Model I TRSDOS will make the copy.

Other TRSDOS will prompt you to switch the source and destination diskettes. Since the "source" and "destination" diskette are the same, simply leave the diskette in the drive and keep hitting ENTER. Dumb machine . . .

You can use the same technique on any other drive.

CHON
Commands, Help On, Model II/III TRSDOS

After TRSDOS READY, type HELP. TRSDOS will respond with the commands on which help is available. The help is somewhat parsimonious — a brief description and the "syntax" (format) of the command.

Now enter

```
HELP command
```

where "command" is the command from the list, for example,

```
HELP FREE
```

Enter

```
HELP SYNTAX
```

to get a description of the format "notation."

CHTP
Columns, How to Put Things In, in BASIC

What to columnate . . . uh . . . columerize . . . uh . . . make nice, neat columns, eh? Here are 37 ways (well, maybe a few less):

1. **Most obvious**: Use the comma symbol on PRINT or LPRINTs, as in

```
100 PRINT A,B,C,D
```

The comma means "tab to the next print zone." The print zones are predefined positions. On the Model I/III, they are at 0, 16, 32 and 48. On the Model II, they are at 0, 14, 28, 42 and 56. On the Color Computer, they are at 0 and 16.

When the data to be printed in each column is of various widths, the column will be printed with a "ragged" right edge and a "justified" left edge, as in

```
1234.5
10000.78
12
1234
```

See below for ways to justify the right edge.

2. Use TAB(XX) command. The TAB command moves the printing (or LPRINTing) to the specified character position. Character positions are numbered from 0 to 63 (Model I/III) or 79 (Model II) or 31 (Color Computer). The following would print items at columns, 10, 20, 30 and 40, with a "ragged right:"

```
100 PRINT TAB(10);A;TAB(20);B;TAB(30);C;TAB(40);D
```

3. Change numeric to string by the STR$ command, test the length and columnate. The trouble with printing numeric variables is that you don't know how many characters will be printed. On numbers with decimal points, trailing zeroes are suppressed, and leading zeroes are not used. It's almost impossible to predict, therefore, where to tab for a numeric variable.

You can get around this by using STR$ to change a numeric variable to a string, as in:

```
100 A=1234.567
110 A$=STR$(A)
120 B=LEN(A$)
```

The numeric variable A is converted to a 9-byte string with intermediate decimal point and a leading blank. The leading blank is used because the number could be a negative number, in which case a minus sign would lead.

The string length can now be measured by the LEN command (see SFLO), and the string can be otherwise manipulated (for example, a dollars and cents value can be made up by adding trailing zero characters). The LEN function can be used in a TAB command to tab to the proper position and to justify the right edge:

```
100 PRINT TAB(20-LEN(A$));A$;TAB(40-LEN(B$));B$
```

4. Use a PRINT USING command to define how many characters should be displayed or printed (not applicable on Level I or Color Computer BASIC). There is more on this in PRUU.

CLCN
Cleaning Connectors

A tried and true method for cleaning "edge connectors" such as you'll find on the Model I/III printer and I/O bus ports: take a large moderately hard eraser, such as a "Pink Pearl" type, and rub it vigorously across the pins of the edge connector. This is fairly effective in removing oxidation from the connector, and doesn't hurt the connector in any way. (Well, possibly after 10,000 passes there might be a modicum of wear . . .) It may be necessary to use a pencil eraser to fit through the opening to reach the connectors.

CLTD
Clearing the Display in BASIC, All Systems

To clear the display while in BASIC, enter CLS (Model I/II) or press CLEAR (Model III, Color Computer).

This action clears the display, deletes any current line, converts to 64 characters per line (Model I/III), and positions the cursor to the upper-left corner (the "HOME" position).

CMD1
Calling a Machine Language Program in Disk BASIC, Model I/III TRSDOS/LDOS

If you have a Model I or Model III with Disk BASIC and want to interface to a machine language program, follow these steps:

1. Assemble or translate the machine language code to be loaded so that it will run properly at the location in RAM memory you desire (see S1EA or EDAN).

2. Protect the memory into which the machine language program is to be loaded by the procedure given in PROT.

3. Put the machine language code into the area of RAM memory required. This can be done by loading the machine language object program with a LOAD command (see LMLD), by POKEing the machine language code (see PPKU), by loading the machine language program by DEBUG (see DT1U) or by other means.

4. Write your BASIC program so that it includes a DEFUSR statement. The format of this is DEFUSRn=XXXXX, where n is a digit of 0 through 9 and XXXXX is the address of the start of the machine language program. Since there are 9 different DEFUSRs, DEFUSR0 through DEFUSR9, you can have 10 separate entry points in the machine language program, 10 separate machine language programs, or a combination of entry points and programs. If one entry point was **8000H** and another **8010H**, for example, you might have:

```
1ØØ DEFUSRØ=32768-65536 'entry point 1
11Ø DEFUSR3=32774-65536 'second entry point
```

Note that the special form in BASIC for addresses over 32767 was used here.

5. Transfer control to the machine language program by performing a USRn call. The format of USRn is A=USRn(B), where n is 0 through 9 (and corresponds to the DEFUSR n digit), A is a "return" variable, and B is an integer variable that can be passed to the machine language routine. The USRn call will take the machine language address defined in the DEFUSRn command and "call" that location, saving the return point of the next BASIC statement after the USR call. If the machine language program requires "arguments," pass them by procedure PVM1. If the machine language program does not require arguments, you may use a USRn function call with a "dummy" argument of 0 - USRn(0) — and a "return" variable that is also a dummy, such as variable A as in 100 A=USRn(0).

Here's an example: We have a simple assembly language program to clear the screen shown in Figure CML1-1. This has been translated to machine code decimal values as given in procedure CFHD, and incorporated into BASIC DATA statements as shown below. The BASIC code to relocate the machine code and to call the machine language program is shown below.

```
1ØØ REM MACHINE CODE IN DATA STATEMENTS
11Ø DATA 62,32,5Ø,Ø,6Ø,33,Ø,6Ø,17,1,6Ø,1,255,3,
    237,176,2Ø1
12Ø REM MOVE THE CODE TO 33ØØØ
13Ø FOR I=33ØØØ TO 33ØØØ+16
14Ø READ A:POKE I-65536,A
15Ø NEXT I
16Ø REM ESTABLISH THE START ADDRESS OF 33ØØØ
17Ø DEFUSRØ=33ØØØ-65536
18Ø REM NOW CALL THE CODE BY A USR CALL
19Ø A=USRØ(Ø)
2ØØ REM RETURN AT THIS POINT
```

Note that the special form of XXXXX-65536 was used in the POKE and DEFUSR above.

CMEC
Calling a Machine Language Program in Extended Color BASIC, Color Computer

If you have a Color Computer with Extended Color BASIC and want to interface to a machine language program, follow these steps:

1. Assemble or translate the machine language code to be loaded so that it will run properly at the location in RAM memory you desire (see EDCE).

2. Protect the memory into which the machine language program is to be loaded by the procedure given in PROT.

3. Put the machine language code into the area of RAM memory required. This can be done by loading a machine language program with a CLOADM command (see LEMC), by POKEing the machine language code (see PPKU) or by the Z-BUG portion of EDTASM+.

4. Write your BASIC program so that it includes a DEFUSR statement. The format of this is DEFUSRn=XXXXX, where n is a digit of 0 through 9 and XXXXX is the address of the start of the machine language program. Since there are 9 different DEFUSRs, DEFUSR0 through DEFUSR9, you can have 10 separate entry points in the machine language program, 10 separate machine language programs, or a combination of entry points and programs. If one entry point was **$3E00** and another **$3F00**, for example, you might have:

```
1ØØ DEFUSRØ=&H3EØØ   'entry point 1
11Ø DEFUSR3=&H3FØØ   'second entry point
```

5. Transfer control to the machine language program by performing a USRn call. The format of USRn is A=USRn(B), where n is 0 through 9 (and corresponds to the DEFUSR n digit), A is a "return" variable and B is an integer variable that can be passed to the machine language routine. The USRn call will take the machine language address defined in the DEFUSRn command and "call" that location, saving the return point of the next BASIC statement after the USR call. If the machine language program requires "arguments," pass them by procedure PVMC. If the machine language program does not require arguments, you may use a USRn function call with a "dummy" argument of 0 - USRn(0) — and a "return" variable that is also a dummy, such as variable A as in 100 A=USR(0).

Here's an example: We have a simple assembly language program to clear the screen as shown in Figure CMLC-1. This has been translated to machine code decimal values as given in procedure CFHD, and incorporated into BASIC DATA statements as shown below. The BASIC code to relocate the machine code and to call the machine language program is shown below.

```
100 REM MACHINE CODE IN DATA STATEMENTS
110 DATA 142,4,0,16,142,2,0,134,32,167,128,49,
    63,38,250,57
120 REM MOVE THE CODE TO $3E00
130 FOR I=&H3E00 TO &H3E0F
140 READ A:POKE I,A
150 NEXT I
160 REM ESTABLISH THE START ADDRESS OF $3E00
170 DEFUSR0=&H3E00
180 REM NOW CALL THE CODE BY A USR CALL
190 A=USR0(0)
200 REM RETURN AT THIS POINT
```

C

CMHT
Clearing Memory, Model II/III TRSDOS, Model I/III LDOS

Model II: Use the CLEAR command in TRSDOS to zero user memory (see MMM2) and return to TRSDOS:

```
CLEAR
```

Model III: Use the CLEAR command to clear all of user memory from **6000H** (MMM1), set memory protect to the end of memory, clear the display and reset all I/O drivers:

```
CLEAR
```

To clear a selected block of user memory, use the following form of the CLEAR command:

```
CLEAR (START=ssss,END=eeee)
```

In this command, ssss is a start address and eeee is an end address, both in 4-digit hexadecimal form (CFDH) with leading zeroes for values greater than **9FFFH**. The end address must be larger than the start, of course, and the start address must be greater than **6000H**.

To set memory protect to a selected address, use:

```
CLEAR (MEM=pppp)
```

where pppp is a 4-digit hexadecimal address (CFDH) without leading zeroes.

Why would you want to use this? This could be used to zero out an area used for a machine language table or other data that must be initialized. CLEAR isn't a necessary prerequisite for BASIC or other system programs, however, and it's somewhat of a cheap frill.

Model I/III LDOS: Use the MEMORY command as follows:

```
MEMORY (CLEAR)
```

This will zero all memory from **5200H** through top of memory or "HIGH$" location. HIGH$ is defined by:

```
MEMORY (HIGH=aaaa)
```

where aaaa is a hexadecimal (see CFDH) memory location.

CML1
Calling a Machine Language Program in Non-Disk BASIC, Model I or Model III

If you have a Model I or Model III without disk BASIC and want to interface to a machine language program, follow these steps:

1. Assemble or translate the machine language code to be loaded so that it will run properly at the location in RAM memory you desire (see S1EA or EDAN).

2. Protect the memory into which the machine language program is to be loaded by the procedure given in PROT.

3. Put the machine language code into the area of RAM memory required. This can be done by loading a machine language program with a SYSTEM command (see LMFN), by POKEing the machine language code (see PPKU), by loading the machine language program by cassette DEBUG or by other means.

4. Write your BASIC program so that it includes the following statements:

```
100 AA=XXXXX
110 POKE 16527,INT(AA/256)
120 POKE 16526,AA-INT(AA/256)*256
```

In the statements above, variable AA is the address of the start of the machine language code. Use the actual numeric value in the POKEs instead if you desire. You may have something like INT(32000/256), for example.

5. Transfer control to the machine language program by performing a USR call. The USR call will take the machine language address in 16526, 16527 and "call" that location, saving the return point of the next BASIC statement after the USR call. If the machine language program requires "arguments," pass them by procedure PVM1. If the machine language program does not require arguments, you may use a USR function call with a "dummy" argument of 0 - USR(0) — and a "return" variable that is also a dummy, such as variable A as in 100 A=USR(0).

Here's an example: We have a simple assembly language program to clear the screen shown in Figure CML1-1. This has been translated to machine code decimal values as given in procedure CFHD, and incorporated into BASIC DATA statements as shown below. The BASIC code to relocate the machine code and to call the machine language program is shown below.

```
1ØØ REM MACHINE CODE IN DATA STATEMENTS
11Ø DATA 62,32,5Ø,Ø,6Ø,33,Ø,6Ø,17,1,6Ø,1,
    255,3,237,176,2Ø1
12Ø REM MOVE THE CODE TO 33ØØØ
13Ø FOR I=33ØØØ TO 33ØØØ+16
14Ø READ A:POKE I-65536,A
15Ø NEXT I
16Ø REM ESTABLISH THE START ADDRESS OF 33ØØØ
17Ø POKE 16527,INT(33ØØØ/256)
18Ø POKE 16526,33ØØØ-INT(33ØØØ/256)*256
19Ø REM NOW CALL THE CODE BY A USR CALL
2ØØ A=USR(Ø)
21Ø REM RETURN AT THIS POINT
```

Note that the special form in BASIC for addresses over 32767 (XXXXX –65536) was used in the POKE above.

Figure CML1-1 – *Clear Screen Program*

```
                  ØØ1ØØ ; CLEAR  SCREEN PROGRAM
ØØØØ  3E2Ø        ØØ11Ø CLRSCN LD     A,' '           ;blank
ØØØ2  32ØØ3C      ØØ12Ø        LD     (3CØØH),A       ;zero first char
ØØØ5  21ØØ3C      ØØ13Ø        LD     HL,3CØØH        ;source
ØØØ8  11Ø13C      ØØ14Ø        LD     DE,3CØ1H        ;destination
ØØØB  Ø1FFØ3      ØØ15Ø        LD     BC,1Ø23         ;1st byte filled
ØØØE  EDBØ        ØØ16Ø        LDIR                   ;chain move
ØØ1Ø  C9          ØØ17Ø        RET                    ;return
ØØØØ              ØØ18Ø        END
ØØØØØ Total errors
```

THESE FIGURES IN DECIMAL ARE:
62,32,5Ø,Ø,6Ø,33,Ø,6Ø,17,1,6Ø,1,255,3,
237,176,2Ø1

CMLC
Calling a Machine Language Program in Color BASIC, Color Computer

If you have a Color Computer with Color BASIC and want to interface it to a machine language program, follow these steps:

1. Assemble or translate the machine language code to be loaded so that it will run properly at the location in RAM memory you desire (see EDCE).

2. Protect the memory into which the machine language program is to be loaded by the procedure given in PROT.

3. Put the machine language code into the area of RAM memory required. This can be done by loading a machine language program with a CLOADM command (see LEMC), by POKEing the machine language code (see PPKU) or by other means.

4. Write your BASIC program so that it includes the following statements:

```
1ØØ AA=XXXXX
11Ø POKE 275,INT(AA/256)
12Ø POKE 276,AA-INT(AA/256)*256
```

In the statements above, variable AA is the address of the start of the machine language code. Use the actual numeric value instead if you desire. You may have something like INT(32000/256), for example.

5. Transfer control to the machine language program by performing a USR call. The USR call will take the machine language address in 275, 276 and "call" that location, saving the return point of the next BASIC statement after the USR call. If the machine language program requires "arguments," pass them by procedure PVMC. If the machine language program does not require arguments, you may use a USR function call with a "dummy" argument of 0 - USR(0) — and a "return" variable that is also a dummy, such as variable A as in 100 A=USR(0).

Here's an example. We have a simple assembly language program to clear the screen as shown in Figure CMLC-1. This has been translated to machine code decimal values as given in procedure CFHD and incorporated into BASIC DATA statements as shown below. The BASIC code to relocate the machine code and to call the machine language program is shown below.

Figure CMLC-1 – *Clear Screen Program*

```
ØØØ99 * CLEAR SCREEN
ØØØØ 8E    Ø4ØØ      ØØ1ØØ CLRSCN  LDX     #$4ØØ    START OF SCREEN
ØØØ3 1Ø8E Ø2ØØ       ØØ11Ø         LDY     #512     FULL SCREEN
ØØØ7 86    2Ø        ØØ12Ø         LDA     #$2Ø     BLANK CHAR
ØØØ9 A7    8Ø        ØØ13Ø CLRØ1Ø  STA     ,X+        STORE BLANK
ØØØB 31    3F        ØØ14Ø         LEAY    -1,Y       COUNT -1
ØØØD 26    FA        ØØ15Ø         BNE     CLRØ1Ø     GO IF NOT 512
ØØØF 39              ØØ16Ø         RTS              RETURN
           ØØØØ      ØØ17Ø         END
ØØØØØ TOTAL ERRORS

CLRØ1Ø  ØØØ9
CLRSCN  ØØØØ
```

```
1ØØ REM MACHINE CODE IN DATA STATEMENTS
11Ø DATA 142,4,Ø,16,142,2,Ø,134,32,167,128,49,63,
    38,25Ø,57
12Ø REM MOVE THE CODE TO 16ØØØ
13Ø FOR I=16ØØØ TO 16ØØØ+15
14Ø READ A:POKE I,A
15Ø NEXT I
16Ø REM ESTABLISH THE START ADDRESS OF 16ØØØ
17Ø POKE 275,INT(16ØØØ/256)
18Ø POKE 276,16ØØØ-INT(16ØØØ/256)*256
19Ø REM NOW CALL THE CODE BY A USR CALL
2ØØ A=USR(Ø)
21Ø REM RETURN AT THIS POINT
```

CNER
CN Error, BASIC

Can't continue error. You can only continue from a STOP or a BREAK, and not from an END or Edit. (The Edit re-shuffles program lines and clobbers variables; hence, you cannot restart with variables intact).

CPCA
Centronics (Printer) Cables, Model I/II/III

The "standard" Radio Shack Centronics cable for connecting to a parallel (see RSWI for description of parallel) printer is shown in Figure CPCA-1. The "Centronics" refers to a printer manufacturer that established a cabling and logic signal structure that became a de facto standard for connections to printers. All Radio Shack parallel and serial/parallel printers use this cable arrangement for parallel data transfer.

You can buy the Radio Shack cables for connecting your Model I, II, or III to a parallel printer (26-1401 or 26-4401), or you can make your own for the Models I and III. The "Centronics" type 36-pin connector is available from electronics parts stores and is usually called a "Centronics" connector (Ampenol 57-30360). The 34-pin "edge card" connector is also available from parts stores or from the Shack (276-1564). Use any "ribbon cable" and strip it down to 36 conductors. See SHTO for information on soldering.

Another alternative: For long cable runs use a "stripped down" version of the cable. The cable connectors intersperse ground wires with "signal" wires for electrical noise immunity. In fact, about half of these ground wires can be eliminated without noise problems, making for a thinner cable that is much easier to string. I have used a "minimum number of wires" cable for several years with no problems on any machine: Model I, II or III. The wiring diagram is shown in Figure CPCA-2.

Figure CPCA-1 – *Parallel Printer Cable*

CENTRONICS CONNECTOR PIN	34-PIN EDGE CONNECTOR PIN	CENTRONICS CONNECTOR PIN	34-PIN EDGE CONNECTOR PIN
1	1	19*	2
2	3	2Ø	4
3	5	21	6
4	7	22	8
5	9	23	1Ø
6	11	24	12
7	13	25	14
8	15	26	16
9	17	27	18
1Ø	19	28	2Ø
11	21	29	22
12	23	3Ø	24
13	25	31	26
14	27	32	28
15	29	33	3Ø
16	31	34	32
17	33	35	34
18*	NONE	36	NONE

* CONNECT PINS 18,19 TOGETHER FOR MODEL II

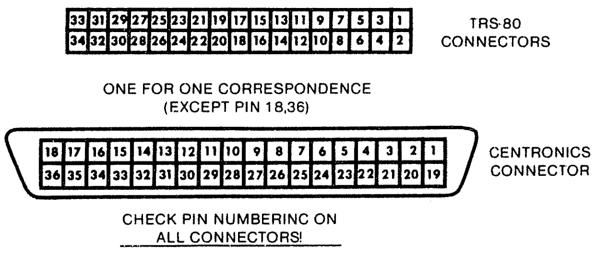
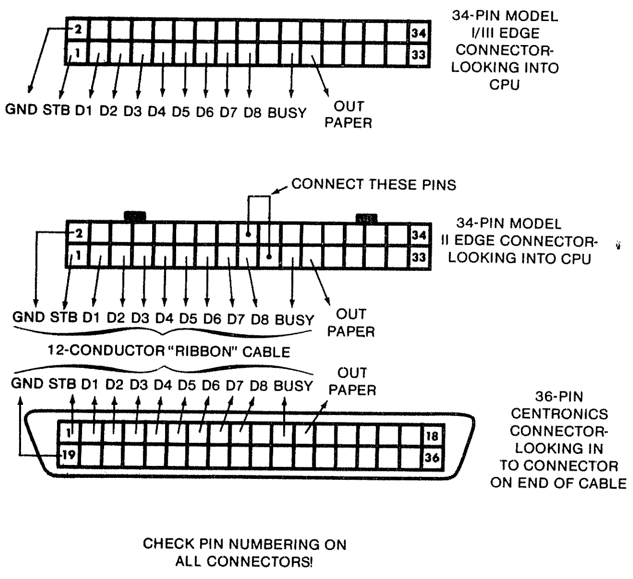


Figure CPCA-2 – *'Stripped-Down' Printer Cable*


2 34
1 33
34-PIN MODEL I/III EDGE CONNECTOR-LOOKING INTO CPU
GND STB D1 D2 D3 D4 D5 D6 D7 D8 BUSY OUT PAPER
CONNECT THESE PINS
34-PIN MODEL II EDGE CONNECTOR-LOOKING INTO CPU
GND STB D1 D2 D3 D4 D5 D6 D7 D8 BUSY OUT PAPER
12-CONDUCTOR "RIBBON" CABLE
GND STB D1 D2 D3 D4 D5 D6 D7 D8 BUSY OUT PAPER
1 18
19 36
36-PIN CENTRONICS CONNECTOR-LOOKING IN TO CONNECTOR ON END OF CABLE
CHECK PIN NUMBERING ON ALL CONNECTORS!

CPM3
Compressing Programs, Model III TRSDOS

The CMD"C" command in the Model III "compresses" BASIC programs by deleting remarks, and or spaces. Compression will make programs run faster in less memory. On the other hand, they will be more difficult to read. Best bet: completely debug your programs first, then compress for speed and compactness. Keep a copy in the uncompressed form for those inevitable changes or bugs that come up months later.

CMD"C" will compress by deleting remarks (CMD"C",R) , by deleting spaces (CMD"C",S), or by deleting both (CMD"C").

When the REMarks option is used, *text* from REMarks and single-quote type REMarks (see QWII) is deleted. The line number and REM remain, however.

When the Spaces option is used, all non-string spaces are deleted from BASIC statements. Any string properly used (double quotes at the beginning *and* end) will be unchanged.

CPSU
CompuServe, Using

CompuServe is the large Data Base in the Sky. It is a data communications system that you can dial up with your Model I, II, III or Color Computer system if you have an RS-232-C driver, the necessary hardware and a modem (see MWAT, MHTU). CompuServe offers hundreds of services — display of news from Associated Press, international, national, and local weather, computer language processors, games such as Space War, programmed instruction, stock market and commodities quotes and a CB radio simulator with which you can talk to other users! If you have never used your RS-232-C and modem, I suggest calling up a local Bulletin Board System first (see BBUS) to gain some experience in data communications and to guarantee that your system is working properly.

To access CompuServe:

1. Get the Radio Shack Videotext software package for your Model I,II,III, or color computer. If you have LDOS or a similar operating system that already has a flexible RS-232-C "driver" and data communications utility, you may only require the "dumb terminal" package. You could use your own simple data communication program except for one small detail — the RS package comes with an application blank for CompuServe, and one hour of free "time." Unfortunately (at this time of writing) you cannot simply sign up without getting the package. In addition, the package contains a rudimentary instruction book for CompuServe.

2. Using the first part of procedure MHTU for a modem, or ACHT for a coupler, hook up your system to a modem.

3. Set your RS-232-C interface to word length of 8, 1 stop bits, no parity, 300 baud, full duplex.

4. Load the Videotext or dumb terminal program.

5. Call the (800) toll-free number in the CompuServe instruction book to find a local number (toll-free) that you can dial up to connect to the central facility.

6. Complete procedure MHTU or ACHT to dial the local number and get a screen display.

7. At this point, you'll see the display shown in Figure CPSU-1. Enter "CIS" or "CPS" to the "Host Name" prompt. Utter the magic words, "the winner for the best marketing effort for a data communication package is..." and enter your user number from the sealed packet. If you have entered your ID correctly, the system will then ask you for your password. Enter the password from the sealed envelope exactly as given. The system will now go on . . . and on . . . and on . . .

8. The options from this point are voluminous. Here are some hints:

   A. If you have a system that allows you to dump the display to the line printer (as the LCOMM driver does in LDOS), do so immediately. You'll have a permanent record of instructions, menus and indices. An alternative would be dumping to disk.

   B. Get a listing of the CompuServe Index. The Index lists all major functions that can be accessed (but there are many, many subfunctions that you do not see). A typical index is shown in Figure CPSU-2.

   C. Use the GO XXX-XX function to go directly to a "page" from the index. This saves time and telephone charges. The GO command can be entered after any "!" prompt, as shown in Figure CPSU-3.

   D. To stop at most times, entering a "control C" will take you back to the last "menu," as shown in Figure CPSU-4.

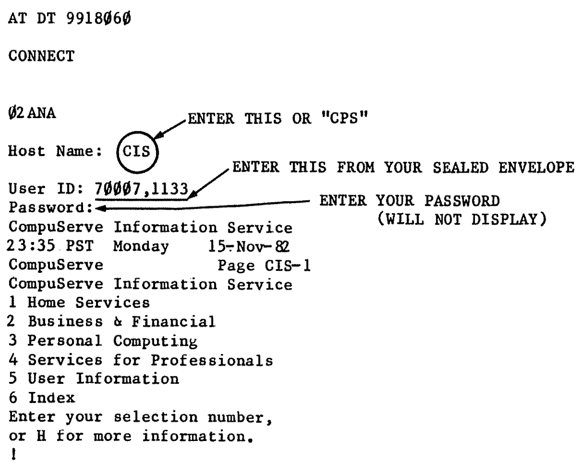
   E. Entering a "T" after a "!" prompt will bring you back to the very first menu, as shown in the figure.

9. You have 1 hour of "free" time. Before that hour runs out, sign up by finding the index entry. You may charge to Visa or Mastercharge or be billed directly. (Frankly, I don't trust computers and prefer direct billing to getting a charge bill for $3003.89 . . .). If you sign up before your hour runs out, you will immediately be given an additional 2 hours that you must pay for. The basic cost at this time of writing is $5.00 per hour — not bad for a local call.

Some negative aspects: I was surprised to learn in first using the system that only the hours from 6:00 pm through 5:00 am are available at the $5 rate. This fact was not in *any* documentation. Prime hours cost $22 per hour. Another peeve: sometimes you'll go from one menu to the next to the next, and finally get down to the topic you're looking for, only to find out there isn't any significant data available (other than sales promotions). Of course, you should expect this from some (ahem!) of the larger microcomputer manufacturers.

By and large, however, the service is excellent, and I hate to use that word.

Figure CPSU-1 – *CompuServe Display*



```
AT DT 9918Ø6Ø

CONNECT

Ø2 ANA
                          ENTER THIS OR "CPS"
Host Name: CIS
                               ENTER THIS FROM YOUR SEALED ENVELOPE
User ID: 7ØØØ7,1133
Password:                                ENTER YOUR PASSWORD
CompuServe Information Service                (WILL NOT DISPLAY)
23:35 PST  Monday    15-Nov-82
CompuServe             Page CIS-1
CompuServe Information Service
1 Home Services
2 Business & Financial
3 Personal Computing
4 Services for Professionals
5 User Information
6 Index
Enter your selection number,
or H for more information.
!
```

Figure CPSU-2 – *CompuServe Index*

```
CompuServe             Page CIS-1
CompuServe Information Service
1 Home Services
2 Business & Financial
3 Personal Computing
4 Services for Professionals
5 User Information
6 Index
Enter your selection number,
or H for more information.
!6◄──────────────────────────SELECTS INDEX

CompuServe             Page IND-1
CIS Subject Index
----------------
1 To Search Index
2 Complete Index List
Last menu page. Key digit
or M for previous menu.
!2◄────────────────────────── SELECTS INDEX LISTING

CompuServe             Page IND-4
 ******************************
             INDEX
 ------------------------------
 COMPUSERVE INFORMATION SERVICE
 ------------------------------
 (Go 54 for quick access to
 page numbers in this index.)
 AAMSI Medical Forum:...Go SFP-5
 AID calculations:.....Go PCS-72
 AMEX prices (MQUOTE):.Go FIN-2Ø
 ASCMD SIG:.............Go SFP-7
 A.S.I. Monitor:.......Go HOM-2Ø
Key S or <ENTER> to continue
!

CompuServe             Page IND-5
 Access:...............Go PCS-3Ø
 Access phone numbers:.....CIS-4
 Adult education:......Go HOM-7Ø
 Adventure game:.......Go GAM-11
 Advertising:
  For sale.............Go HOM-3Ø
  Notices..............Go HOM-3Ø
  Want ads.............Go HOM-3Ø
 Advertising, classified:
  S.L. Post-Dispatch.Go SPD-1ØØ2
 Advice:
  Aunt Nettie.............Go NET
Key S or <ENTER> to continue
!

CompuServe             Page IND-6
 African weather:......Go CNS-17
 Agricultural news:....Go CNS-14
 Aircraft:................Go ASI
 Aircraft (SIG):........Go SFP-6
 Air Travel:..............Go PAN
```

Figure CPSU-3 – *CompuServe 'Go' Prompt*

```
 Boston, Shawmut Bank:....Go SHW
 Bridge game:..........Go GAM-18
 Broadcasting:............Go NPR
 Brokerage:...............Go UMC
 Budgeting, home:
  Gov't publications....Go GPO-4
  CIS calculations.....Go HOM-8Ø
 Bulletin board:.......Go HOM-3Ø
 Business ^C
Key S or <ENTER> to continue
!g pcs-5Ø◄──────────────────────── "GOES" TO "PAGE"
                                        PCS-50

CompuServe             Page PCS-5Ø
COMPUTER GROUPS AND CLUBS
 1 CP/M Group      9 MNET8Ø TRS8Ø
 2 HUG  (Heath) 1Ø LDOS   TRS8Ø
 3 MAUG (Apple) 11 VTOS   ST8Ø
 4 MNET-11(H11) 12 TeleComm Now
 5 MUSUS=PASCAL 13 CLUBIG
 6 RCA  Group   14 AUTHOR'S SIG
 7 TRS8Ø Color  15 Commodore
 8 Panasonic    16 Atari Forum
17 Instructions 18 Descriptions
Last menu page. Key digit
or M for previous menu.
!
```

Figure CPSU-4 – *CompuServe Control C*

```
 #: 1Ø663      Sec. 1 - CC Hardware
Sb: POKE 65495,Ø
    Ø9-Nov-82  18:29:Ø7
Fm: CHARLES 79999,999
To: Wayne Night 78888,8888 (X)
NO. FORGOT BOUT THAT. TNX. WILL DO.
 #: 1Ø665      Sec. 1 - CC Hardware
Sb: #Reverse Screen Mod
    Ø9-Nov-82  18:33:24
Fm: CHARLES 79999,999
To: STEVE SMITH 77777,777 (X)
HAVE A 32K W/VIDEOTEX ROMPAK. UNABLE TO FIGURE WHICH BOARD IT
IS 'CAUSE ALL I CAN SEE THRU THE VENTS IS THE NUMBER ON THE
FEMALE SIDE OF THE ROM PLUG AND THERE'S NO 'E' OR 'D' OR
NOTHING. HAVEN'T TRIED DEBUGGING THE THING YET - JUST SAW 2
VARIATIONS ON A THEME AND TRIED BOTH. THE GRNSCN WORKED AND THE REVERSE DIDN'T.
 #: 1Ø666      Sec. 2 - CC Software
Sb: #DATABASE
    Ø9-Nov-82  18:35:Ø1
Fm: CHARLES 79999,9999
To: STEVE SMITH 77777,777 (X)
SURE WOULD APPRECIATE THAT KIND OF PGM. IF U HAVE IT ALREADY
PUT TOGETHER, WUD SAVE A LOT OF TIME. TNX
 #: 1Ø667      S^C  <-----------------------------  "CONTROL" C INTERRUPTS
Enter blank line to continue:                       AT ANY TIME
* The Color SIG *
Function menu:
1 (L)  Leave a message
2 (R)  Read messages
3 (RN) Read new messages
5 (B)  Read bulletins
6 (CO) Online conference
9 (OP) Change your SIG options
Ø (E)  Exit from this SIG
Enter selection or H for help: t <------- ENTERING "T" SELECTS
Exiting at 15-Nov-82  23:41:33            "TOP" MENU
Last message on system: 1112Ø
High message retrieved: 1Ø667
Thank you for visiting * The Color SIG *
CompuServe           Page CIS-1  }
CompuServe Information Service   }
1 Home Services                  }
2 Business & Financial           }
3 Personal Computing             }
4 Services for Professionals     }  TOP MENU
5.User Information               }
6 Index                          }
Enter your selection number,     }
or H for more information.       }
!                                }
```

CPWR
Conditioning Power

If you live in a "noisy" power line environment and share power circuits with heavy equipment or other dwellings, as in a condominium or apartment complex, you may have some problems with power line "glitches" that zap your system. Try these remedies:

1. If you have a Model I, make certain that all of the engineering changes have been installed relating to the expansion interface dynamic memories. Early Model I's should have the so-called "pregnant cable," a cable with an integral electronics box connecting the CPU and expansion interface. Symptoms of noisy environments and dynamic memory problems are "disk reboots" when other equipment on the line switches on, or even at unpredictable times. (My Model I, for example, reboots every time I turn on a fluorescent light on the computer table — my Model III, on the same circuit, is unaffected. Such is progress . . .)

2. Try to isolate your equipment as much as possible from circuits that carry heavy loads, especially large ac motors.

3. Make certain that all of your equipment is properly grounded with a standard three-conductor plug (see ACPR). If certain peripherals have only two plugs, you can buy a special two-conductor to three-conductor grounding plug at any hardware store. Connect the ground wire to a metal screw that connects to the chassis of the equipment, which is usually both a signal and power ground.

4. Get an ac outlet box with three-conductor outlets and power line filtering (RS 26-1451 or various models from Electronic Specialists, Natick, MA).

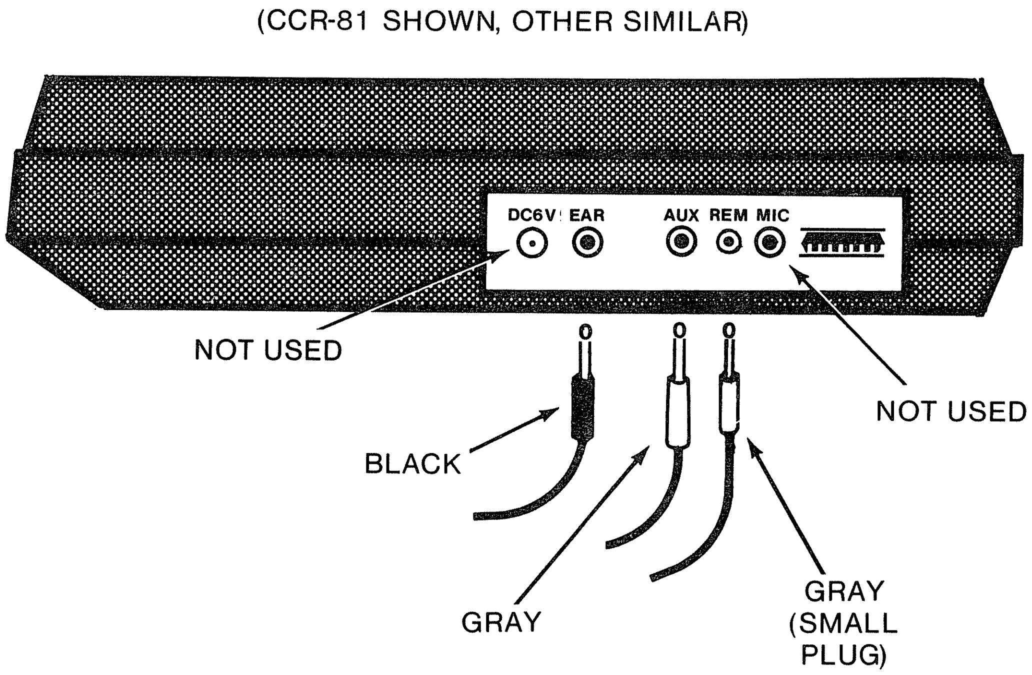
If the above suggestions do not help, consider model railroading.

CRPI
Cassette Recorder Plug, Insertion Of

Figure CRPI-1 shows the correct insertion of cassette recorder plugs on the typical cassette recorders for the I, II, III and CC.

A good tip: Remember the mnemonic device "black is back"; the large black plug goes into the jack farthest to the rear of most recorders.

Figure CRPI-1 – *Cassette Recorder Plugs*



CSFT
Clearing the Screen From TRSDOS, Model II/III

Use the CLS command

```
CLS
```

CSHT
Comparing Strings in BASIC

How do you compare strings in BASIC? Use the conditional operators <, <=, <>, >=, > and = (less than, less than or equal, not equal, greater than or equal, greater than, and equal) and the IF... THEN... statement, as in

```
100 IF A$<=B$ THEN B$=A$
```

The meaning of A$=B$ or A$<>B$ is clear enough, but what about the other relationships? What is a string that is "less than" another? Comparisions of strings are done on a character by character basis, based on their actual numeric value. Character data is normally in ASCII (see ADFW), but the string values may be any value from 0 to 255 (see CHR$ use in CUSE).

If any character in the first string is less than the character in the same position in the second string, then the entire first string is "less than" the second string. If any character in the first string is "greater than" the character in the same position in the second string, then the entire first string is "greater than" the second string. If the strings are of unequal length but contain the same characters up to the length of the smaller, the shorter string is "less than" the other. Examples:

```
"AA" compared to "AAA"              "AA" < "AAA"
"AAA" compared to "AAB"             "AAA" < "AAB"
"AAA" compared to "aaa"             "AAA" < "aaa" !!
"THE QUICK BROWN ARMADILLO LEAPT" compared to "TI"
                                    "THE QU..." < "TI"
"AAA" compared to "aaa"             "AAA" <> "aaa"
```

CSNV
Converting BASIC Strings to Numeric and Vice Versa

There are a number of BASIC commands that enable you to convert string variables to numeric variables and vice versa. These are applicable to all systems except Model I/III Level I systems.

One good reason you might want to convert a numeric value to a string is to make it easier to columnate the value when it's printed or displayed (see CHTP). After a conversion of numeric to string you can use the LEN command to find out how many characters are in the value, and you can even delete or add additional characters by string processing commands (see SACW).

Sometimes it's convenient to do the reverse, to change strings to numeric. A good example of this is changing a date in MM/DD/YY format or a YYYY value into a numeric value for processing. See also "PRINT USING" in PRUU for a convenient way to format numeric displays or "LPRINT USING" to format printouts.

To change a numeric value to a string, use the STR$ function in BASIC. This command converts any numeric variable to a string. Suppose that variable X was equal to -56.789. You could convert to a string by

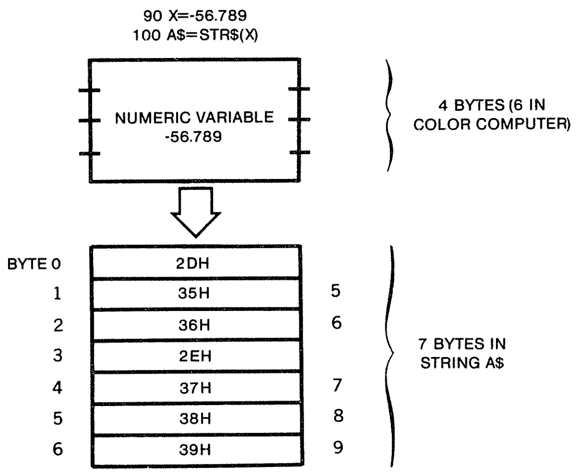
```
100 A$=STR$(X)
```

The resulting string is shown in Figure CSNV-1. The string would consist of seven bytes (LEN=7) — the first byte would be a sign (**2DH**), followed by 5 (**35H**), 6 (**36H**), decimal point (**2EH**), 7 (**37H**), 8 (**38H**), and 9 (**39H**). If the numeric value had been positive, the first byte would have been a blank (**20H**). Here are some rules for numeric to string conversions:

1. There is a leading minus sign in the result string if the numeric value is negative.

2. There is a leading blank in the result string if the numeric value is positive.

Figure CSNV-1 – *Numeric to String Example*



```
90 X=-56.789
100 A$=STR$(X)
```

3. There are no trailing blanks.

4. Decimal points appear as an ASCII decimal point in the proper place.

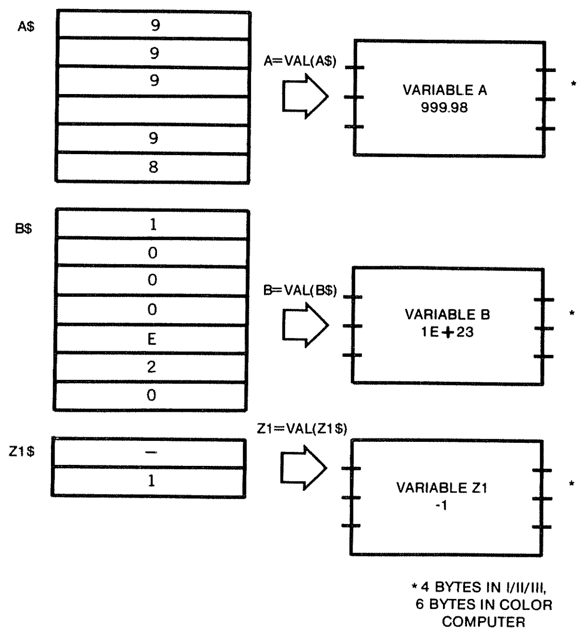
5. Trailing zeroes for a fraction are not present in the result string.

6. Leading zeros are not present in the result string.

To sum it all up, the result is exactly the way the numeric value would appear on a display, with the exception that no trailing blank is generated.

To change a string to a numeric value, use the VAL function in BASIC. This function is the reverse of STR$. It takes a string variable value and converts it to a numeric variable. The strings "999.98", "1000E20" and "-1" would be converted to the corresponding three numeric values shown in Figure CSNV-2.

Figure CSNV-2 – *String to Numeric Example*



Here are some rules for VAL:

1. If the string contains no numeric digits, or if the string is "" (null, no length), a zero is returned.

2. If the string contains no decimal point, the result can be set equal to an integer variable if the string is between -32768 and +32767, except for the Color Computer, which has only one data type.

3. If the string contains a decimal point, the result must be a single-precision or double-precision number, depending upon the number of digits in the string, except for the Color Computer, which has only one data type.

4. If the string has non-digit characters (for example, $ or %), they are ignored and do not affect the result.

CSUM
Clock Speed Up Modifications

If you add one of the available clock speed-up modifications to your Model I, III or Color Computer will you get twice the speed? Yes. BASIC programs will operate twice as fast if the clock speed-up modifications double the basic CPU clock speed.

Do I recommend them? No, for a number of reasons. First, they void your warranty. Secondly, the new computer speed may be too fast for critical system timing such as disk operation, or it may be incompatible with system timing such as software cassette timing or the real-time-clock. Thirdly, the new speed may be too fast for proper chip operation — not so much the cpu chip, which will probably run at higher speeds, but certain types of memories. Fourthly, they require some "hardware" experience for installation.

The opposite side: some users that I know of have made the modifications and have experienced little trouble. Do the mods only if you have hardware construction experience and are prepared to live without Radio Shack service.

CTLC
Cassette Tape Loading Difficulties, Models I/III/Color Computer

If you have difficulties loading a cassette tape file, try these cures:

1. Make certain the tape is positioned properly by removing the EAR output plug and REM plug and listening to the file. Position the tape right before the file and try again.

2. Temporarily remove the AUX input from the cassette recorder and try again.

3. Check the volume control setting. Try various levels.

4. Physically relocate the cassette recorder away from the television or monitor as far as possible. Try either side of the television or monitor; there may be interference from the flyback transformer or other electronics.

5. Are you using a high-quality tape? It may pay to get a "certified" cassette tape such as Radio Shack's 26-301 or 26-302. Certified tapes have been tested to ensure that there are no "dropouts" - points on the tape where the magnetic material is thin or non-existent.

Recommended volume setting for Model I/III/Color Computer: 7 on scale of 10.

Model I users only: If you have an early Model I, you may require a modification called the "XRX-III mod". This was an electronic modification to the cassette circuitry making the cassette input more reliable. Check with your closest service center to see if they are still installing it (Service Information Bulletin #1130).

CUSE
CHR$ Use, BASIC, Most Systems

The CHR$ function creates a one-character string from a numeric value. Unlike generating a literal string (A$="HELP!"), this string may consist of "non-ASCII" (see ADFW) characters. CHR$ is a way of incorporating "control codes," graphics codes, and other non-ASCII characters into strings.

The result string from

```
100 A$="LINE 1"+CHR$(13)+"LINE 2"
```

would be a 13-character string of "LINE 1," a carriage return, and "LINE 2."

The value within the CHR$() may be any value from 0 through 255. See GHS1 for use of CHR$ in Model I/III graphics.

CWTC
Commands, What TRSDOS/LDOS Commands Are There?, Model I/II/III

After TRSDOS READY, type LIB. This action will display all LIBrary commands.

Library commands are simply the repertoire of commands available on the system. Model I/III LDOS divides the commands into Library "A" (generally essential) and Library "B" (may be PURGEd if not required) commands.

D1TW
DEVICE, Model I TRSDOS, What Is It?

DEVICE lists currently defined system devices — KI for keyboard, DO for video display and PR for line printer. As it stands it is a "hook" that was never embellished. DEVICE should show the current status of each system device and other system configuration information.

DADF
Deleting Many Disk Files, How to, Model II/III TRSDOS

Want to delete all user disk files, or many user disk files? Use the PURGE command in TRSDOS. Using PURGE or PURGE :n, where n is the disk drive number 0, 1, 2 or 3, will delete all user files.

Model III: After PURGE is entered, TRSDOS will ask for the master password of the diskette (see PWDS). TRSDOS will then find each user file and ask

```
PURGE FILE Y OR N?
```

This dialog will continue until every user file has been purged from the diskette. You can delete all user files or as many as you require by answering Y or N.

If you want to delete "invisible" files (see VAIF), you can enter

```
PURGE :n (INV)
```

and go through the same dialogue.To delete both "visible" and "invisible" files enter

```
PURGE :n (INV,VIS)
```

To delete *all* files, including System files (see DDWA), you can use

```
PURGE :n (ALL)
```

This command will create a Data Diskette (see DDWA) without any dialogue. A better way to do this is to FORMAT the diskette, as you'll get the advantage of some "certification" of the diskette — flawed areas of the diskette will be found, at the expense of a longer time.

Model II: There are three classes of files that can be purged, SYS (TRSDOS system files), PROG (user machine language files), DATA (user data files). The easiest PURGE is

```
PURGE :n
```

where :n is the drive number (optional for drive 0). PURGE will ask for the disk master password and then display **user** file names one at a time; you can kill any number by answering Y to the prompt.

Use the other options in any combination.

```
PURGE :1 PROG,DATA
```

will enable you to kill all user files.

A special option, ALL, lets you PURGE all files

```
PURGE :2 ALL
```

By selectively PURGEing SYS files, you can create a minimum configuration diskette. PURGE all files except SYSTEM/SYS and SYSTEM32 or SYSTEM64 (depending on your memory option). Note that this is not a data diskette, but is a minimum function diskette that may be used in drive 0.

DCJS
Date Conversion in BASIC, Julian/Standard Format, Model III TRSDOS

The CMD"J" function (boy, these Model III commands never stop, do they — let's see, where's the one for amortizing a loan for a Model II . . .) converts between two date formats.

The first format is the standard MM/DD/YY format that we all use and love so well.

The second format is one useful to programmers that are calculating elapsed time. The "Julian" day is the "elapsed" day of the year, from 0 through 365 or 366. The format used here is YY/DDD, where YY is the last two digits of the year, and JJJ is the Julian number.

To convert from MM/DD/YY format to Julian format, execute

```
CMD"J","MM/DD/YY",string
```

where "MM/DD/YY" is the standard date format, either as a "literal" with quotes around it, or as a string variable. The second string is the result after the conversion.

The command

```
CMD"J","1Ø/24/82",A$
```

results in A$="297"; note that the year is not present.

To convert from Julian format to standard format, execute the command

```
CMD"J","-YY/DDD",string
```

where "-YY/DDD" is a literal string with a leading minus sign, two year digits, and the Julian day number. This string can also be a string variable without the quotes. The result is a string variable.

```
CMD"J","-82/297",A$ produces A$="10/24/82".
```

Julian, by the way, was not Caesar, as many would erroneously say, but Seymour Julian, an early buyer at One Tandy.

DDER
DD Error

Doubly-dimensioned array. You can only establish an array once, and cannot do a DIM statement more than one time. Put all of your DIM statements at the beginning of the program as good programming practice.

Model II owners: Use ERASE to deallocate one or more arrays.

```
ERASE A$,B$
```

releases the A$ and B$ arrays back to the available user RAM space.

D

DDWA
Data Diskettes, What are They?

Data diskettes do not contain any operating system software (see SDWA). They are effectively blank and contain only a disk directory and "bootstrap" loader. A data diskette can never be used in drive 0 — a "system" diskette must be used instead. Data diskettes can be used in any other drive, however, as long as there is a system diskette in drive 0. (LDOS allows exceptions to this for certain short "utility" functions). Data diskettes have almost all of the disk space available for user programs and data. Data diskettes are created by "formatting."

DEHM
Disk Errors, How Many are Too Many?

This is a toughie. It's one thing to read in disk drive or diskette manufacturer's literature about "number of hard errors" per millions or billions of bits and translate that into pragmatic terms. However, never fearing to tread, let me take a stab at it:

For properly calibrated disk drives, hardware with no bugs, and reasonable handling of reliable diskettes, you may experience a "non-recoverable", or "hard" error every 10 diskettes full of data or so. By this I mean that in normal storage of programs and data you've used up 10 diskettes and that no diskette has been subjected to hundreds of reads and writes.

Things that will influence the number of disk errors:

1. If disk drives are not properly aligned, you may find that you cannot switch a diskette from one drive to another without "parity errors," or that you cannot load diskettes generated on other machines. Solution: take the drives into Radio Shack or an independent repair facility to have them aligned. Typical cost about $30-$40 per drive.

2. If there are hardware errors in the disk drive controller, you may have occasional disk errors. The hardware malfunction may be related to heat (or cold). Try to observe whether the errors occur more frequently during temperature extremes or immediately upon power up.

3. Improper handling of diskettes. It's not just diskette manufacturer's propaganda, grease from fingers (sorry, I don't mean to use an argumentum ad hominem) or other substances can effect disk errors. Very important: keep diskettes away from dust by covering your system with a cover and by storing diskettes in a plastic envelope that covers most of the diskette. And of course, avoid blowtorches, solar energy concentrators, Tesla coils and the like.

4. Don't buy cheap diskettes. When your computer club sells those 50 diskettes for $50, are you truly getting a bargain? Not if you lose hours' worth of programs or data. You can get good, reliable diskettes for about $35 or less for ten 5 1/4" diskettes and $50 for ten 8" diskettes. Isn't that inexpensive enough?

5. Brand of diskettes: I'm not going to recommend Verbatim diskettes, or 3M or Dysan. Computer users are very chauvinistic about which diskettes work for them, and as soon as I say Albatross diskettes, I'll get thousands of users that hate them! My recommendation: try several reliable brands and stick with the brand that works best for you. But I like Maxells . . .

6. Using more stringent hardware on single-density machines. If you have a Model I and are using a "foreign" disk drive with 40 tracks, you may be getting disk errors of tracks 35 through 39. Data is packed tighter there (a track has a smaller circumference, but the number of "bit times" is the same), and the "data separator" that separates the clock pulse from data pulses has a tougher job. You may want to add an "external data separator" sold by several "foreign" manufacturers. You may also experience problems using some of the foreign "double-density" add-ons for the Model I; some of these problems are insoluble. What's that about a sow's ear . . .

To resolve disk problems, get a good disk diagnostic. One of the best is Stambaugh's "Floppy Doctor", which includes every disk test imaginable, including interchangeability of diskettes.

DFAH
Disk File Access, How to Speed Up

If your files have been created on a relatively "full" diskette, the files may be "segmented" throughout the disk instead of being in one contiguous area. This makes for longer file accesses. Copy the appropriate files onto a new "clean" diskette, and you'll get faster access times.

See also CREATE (CDFH).

DFBH
DIR from Inside Disk BASIC, How to Do It, Model III TRSDOS

Execute

```
CMD"D:d"
```

where d is the drive number, 0 through 3. Only unprotected, visible files (see FNMH and VAIF) will be displayed.

DFDA
Disk File, Deleting, All Systems

Use the bellicose KILL command. The format is:

```
KILL name
```

where "name" is a standard filename (see FNMH) with extension, if any.

Not applicable to Color Computer: You may need to know the password. If you don't know the password, try a PURGE, on some systems.

Model II users only: You can KILL multiple files that have similar names or extensions. To kill all files with the same extension, use:

```
KILL */ext
```

where "ext" is the extension of the files that you want to KILL. You might have

```
KILL */FEB
```

for example, to KILL all files that have the /FEB extension.

To KILL all files with similar names, use the asterisk to replace a string of one or more letters. To kill all files starting with ACCTS (ACCTS10, ACCTS11, ACCTS12) and with any extension, for example, use:

```
KILL ACCTS*/*
```

Inside Disk BASIC: Use the Disk BASIC command

```
KILL "name"
```

to kill a file from within Disk BASIC where "name" is a valid file name. This command is handy for creating space on a disk without losing a BASIC program in RAM.

DFOT
Date, Finding Out Today's, Model I/II/III Disk Systems

Method One: use a Calendar.

Method Two: use the TRSDOS or LDOS DATE command (DSDS) to display the date you put in earlier in the day (don't rely on the system as an accurate timekeeper).

Enter DATE without any arguments, and the current date will be displayed on the screen.

Model II: Use the DATE$ command (see DIBP).

DI13
Discrete Inputs for the Model I or III

You can use the Model I or III to read remote "real-world" inputs quite easily. Real-world inputs are any inputs that are external to the computer system, such as burglar alarms, fire detectors, and even such things as magnetic "reed" switches closed by magnets. In the following schemes, we're talking about remote switches with slowly changing inputs; the length of the "run" may be 50 feet or more. The cable used can be any garden variety cable from Radio Shack; since the currents are low, you may use speaker cable, "intercom" (3-wire) cable or a similar type of cable.

Method 1: see DICC, Method 1 for schematic and connections for cassette input for one "channel." Read the channel by

```
100 A=PEEK(INP(255) AND 1)
```

Variable A will be a 1 if the switch is on and a 0 if the switch is off.

Method 2: connect two 6-volt batteries and a "single-pole double-throw" switch as shown in Figure DI13-1. Duplicate the connections for up to four switches as shown in the figure. The voltage of the batteries does not have to be exact, but try for around 6 volts. Wire an RS-232-C male plug as shown in the figure (see also RSWI). Read the state of the four switches by

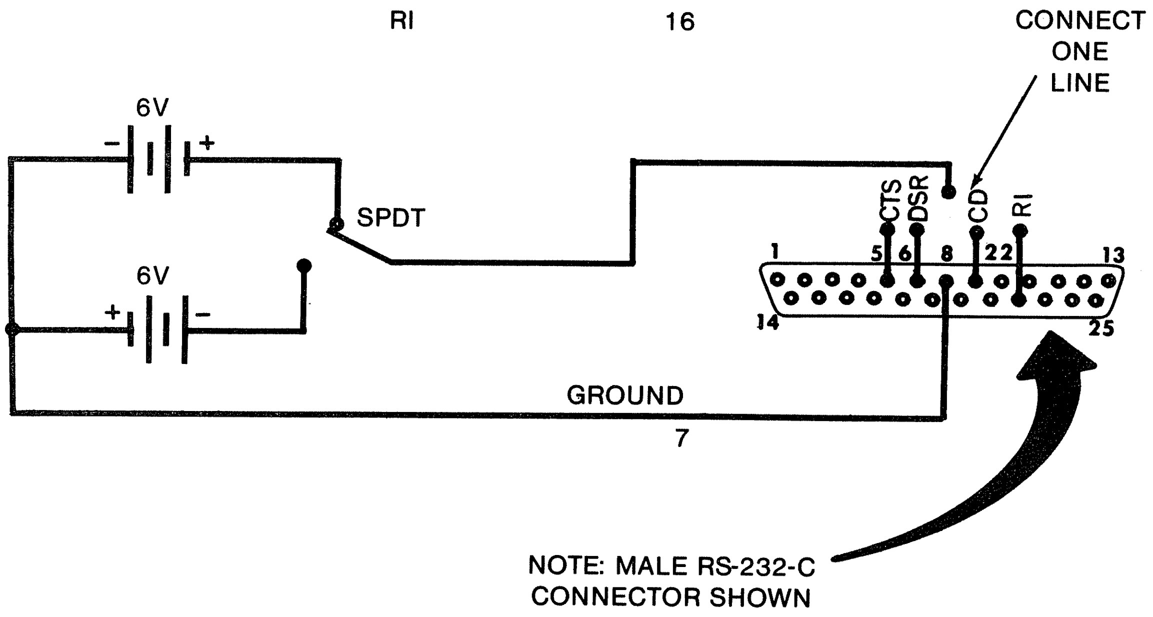
```
100 OUT 232,0
110 A=INP(232) AND 128
120 B=INP(232) AND 64
130 C=INP(232) AND 32
140 D=INP(232) AND 16
```

The switch will be on if the variable is a 1, or off if a variable is a 0. Of course, this method will not work if you also require the RS-232-C port for your printer or modem.

There are several other methods of reading "discrete" inputs on the Model I and III, but these are the easiest. See my Howard W. Sams book "TRS-80 Model I, III, and Color Computer Interfacing Projects" for further information.

Figure DI13-1 – *Model I/III Discrete Inputs*

CONNECTION	AND VALUE
CTS	128
DSR	64
CD	32
RI	16

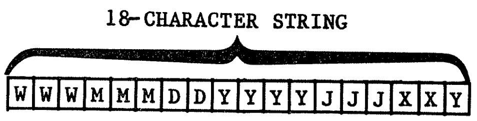


DIBP
Date, in BASIC Program, Model II

The DATE$ function gets the date in the format WWWMMMDDYYYYJJJXXY, where WWW is the day of the week (MON, TUE, WED, THU, FRI, SAT, SUN), MMM is the month (JAN, FEB, MAR, APR, MAY, JUN, JLY, AUG, SEP, OCT, NOV, DEC), DD is the day of the month (1-31), YYYY is the year (19YY), JJJ is the Julian day (see DCJS), XX is the numeric month of the year (1-12) and Y is the numeric day of the week. Monday is day 0. Typical responses are shown in Figure DIBP-1.

Of course, Model II BASIC is not clairvoyant; the correct date and time must have been entered on system start up or the time input by the TRSDOS TIME command (see TSRT).

Figure DIBP-1 – *DATE$ Function Format*



```
Thu  Nov  1 1 1 9 8 2 3 1 5.1 1 3  (11/11/1982)
Tue  Jan    1 1 9 0 1     1   1 1  (1/1/1901)
```

DICC
Discrete Inputs for the Color Computer

You can use the Color Computer to read remote "real-world" inputs quite easily. Real-world inputs are any inputs that are external to the computer system, such as burglar alarms, fire detectors, and even such things as magnetic "reed" switches closed by magnets. In the following schemes, we're talking about remote switches with slowly changing inputs (slower than a dozen times per second or so); the length of the "run" may be 50 feet or more. The cable used can be any garden variety from Radio Shack; since the currents are low, you may use speaker cable, "intercom" (3-wire) cable or a similar type of cable.

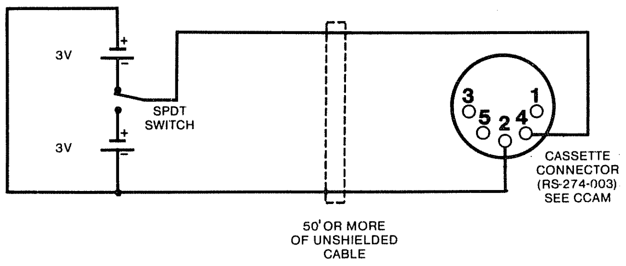
Method 1: connect a "single-pole, double-throw" (SPDT) switch and two three-volt battery supplies as shown in Figure DICC-1. The battery supply may be located at either end of the cable. The Color Computer end of the cable connects to pins 2 and 4 of a cassette plug (see CCAM). You can read the state (on or off) of the switch by doing

```
1ØØ A=PEEK(&HFF2Ø) AND 1
```

Variable A will be a 1 when the switch is on and a 0 when the switch is off.

Method 2: connect one or two "single-pole, single-throw" (SPST) switches and two resistors to each joystick plug, as shown in Figure DICC-2 (see also JPPO). The resistors are plain 10% carbon resistors available from Radio Shack. You can now read each joystick channel (see ADIC) by a JOYSTK command. If the switch associated

Figure DICC-1 – *Color Computer Discrete Inputs I*



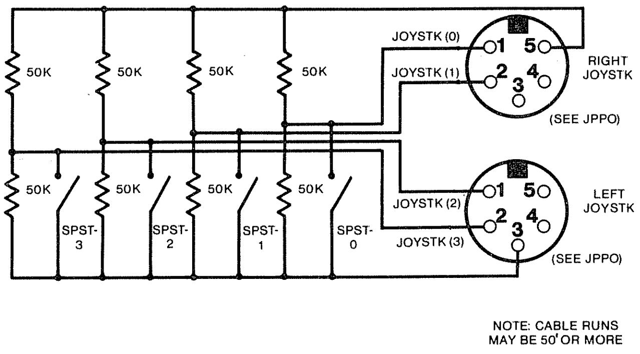
with the channel is open, the value read will be about 32; if the switch is closed, the value read will be about 0. Do this

```
1ØØ IF JOYSTK(Ø)<2Ø THEN PRINT "ON" ELSE PRINT "OFF"
```

Of course, do JOYSTK(1), (2), or (3), depending upon the channel, and make certain that you do a JOYSTK(0) before any of the other JOYSTK commands.

There are several other methods that you can use, but I don't recommend them. One reads the RS-232-C "RD" input in a scheme similar to 1) above, but requires two 6-volt batteries. Another simulates the joystick switch inputs; as the joystick switch inputs are also keyboard inputs, this gets somewhat confusing if you're trying to use the keyboard and monitor remote inputs at the same time. For further information, see my Howard W. Sams book "TRS-80 Model I, III, and Color Computer Interfacing Projects."

Figure DICC-2 – *Color Computer Discrete Inputs II*



DIDI
Directory, TRSDOS/LDOS, All Systems

DIR is used to display directory files on one or more diskettes. Each diskette has a directory on it; the directory is a record of all disk files, the file type and attributes, the file location on disk, file length and other parameters (see ADFC, ADFL).

In the Model I and III, user files are either "visible" or "invisible" (see VAIF). Visible files will list in a simple directory listing. In the Model II and Color Computer, all user files are "visible".

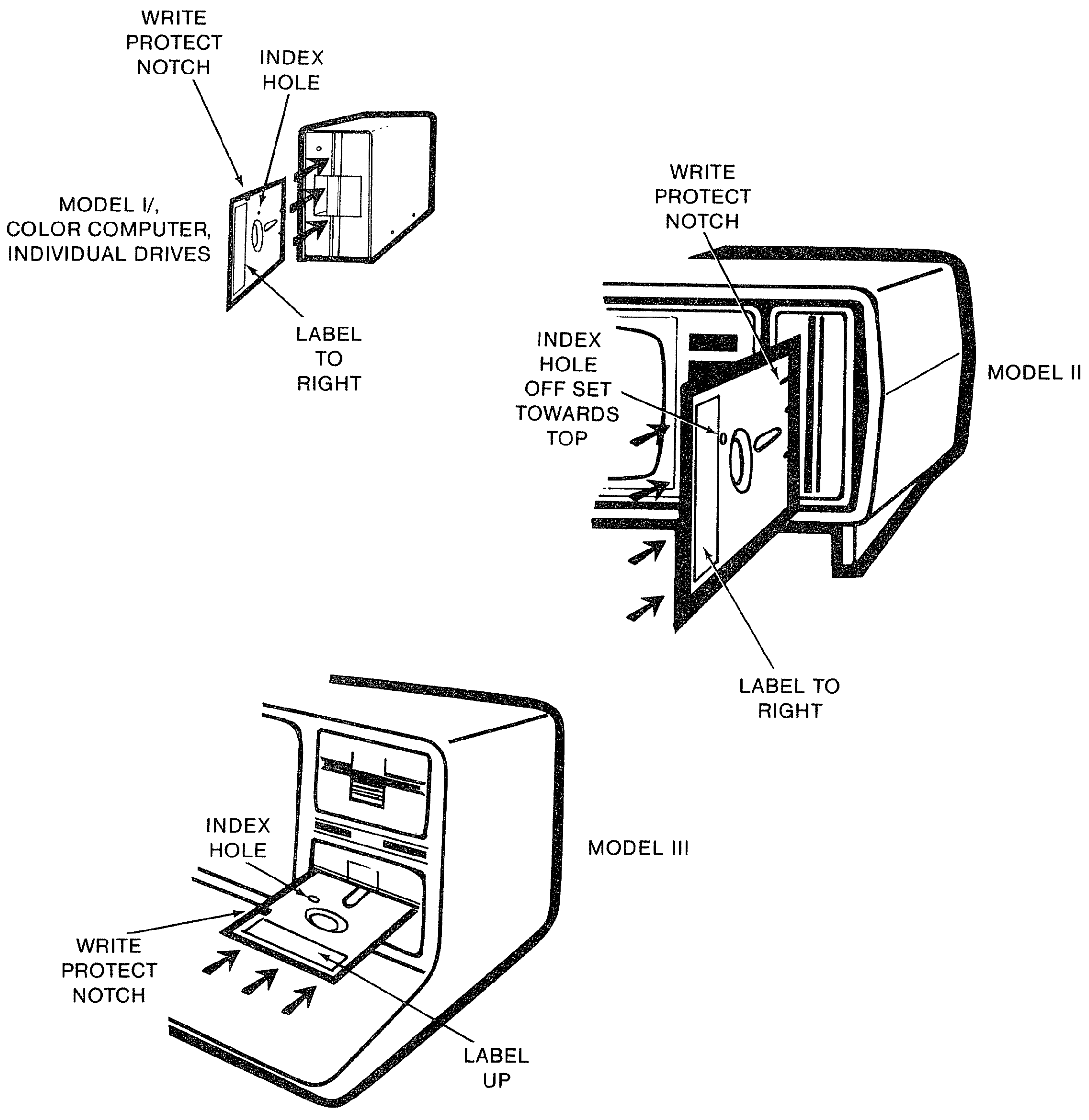
There are also "system" files on the diskette, unless the diskette is a "data" diskette (see DDWA). System files are the modules that make up either TRSDOS or utility programs. System files are also "invisible," but can be listed by a special DIR option.

The directory is usually physically located near the center of the diskette, to minimize disk "search time," as the directory is frequently accessed.

DINS
Diskettes, Inserting

Figure DINS-1 shows the orientation for the Model I, II, III and CC. Cutout is always inserted into the drive. Label is always on top or right side. If there is no label, sector index hole should be towards left or top. We're speaking of single-sided drives only here, of course. Some non-Radio Shack drives are double-sided, in which case either side will work.

Figure DINS-1 – *Inserting Diskettes*



DLDB
DEBUG, Loading from Disk BASIC, Model I TRSDOS, Model I/III LDOS

Execute

```
CMD"D"
```

to load DEBUG from disk. After DEBUG has been loaded, pressing BREAK will transfer control to DEBUG.

DEBUG can be used to examine memory data and machine language programs. Loading DEBUG will clobber BASIC, so it's one of those weird, marginally useful commands.

CMD"D" in Model III TRSDOS displays the directory from inside BASIC. Nothing like consistency.

DLIB
Deleting Lines in BASIC, All Systems

For a single line: type in line number alone. Line will be deleted.

For multiple lines: use DELETE in Model I/II/III (except Level I) and DEL for Color Computer (except Color BASIC). The usual format is

```
DELETE 1ØØ-5ØØ
```

which deletes lines 100 through 500, unless the starting line number doesn't exist.

Use also "-" for start or end line number as in

```
DEL 1ØØ-
```

which deletes lines 100 through end in Color Computer (DELETE line # - not valid on I/III).

DLM1
Directory Listing, Model I, TRSDOS

Read DIDI if you are not familiar with directory files.

To get a directory listing of all visible (user) files, do:

```
DIR :n
```

where :n is the disk drive number (:0-:4). The drive number is optional. If the drive number is omitted, drive 0 is assumed. A "P" opposite a file name indicates that the file has a non-blank password (see FNMH).

To get a directory listing of all visible and invisible user files, do:

```
DIR :d (I)
```

To get a directory listing of all visible and invisible user files and system files, do:

```
DIR :d (I,S)
```

To get a listing of how much space is used for user files, do:

```
DIR :d (A)
```

The resulting listing is shown in Figure DLM1-1. To convert "GRANS" to bytes, see DSHL.

Figure DLM1-1 – *Directory Listing, Model I TRSDOS*

```
FILE DIRECTORY     DRIVE Ø    TRSDOS     1Ø/29/8Ø

BOOT/SYS SIP       LRL= 256 / EOF=   5 / SIZE=    2 GRANS
DIR/SYS SIP        LRL= 256 / EOF=  1Ø / SIZE=    3 GRANS
SYSØ/SYS SIP       LRL= 256 / EOF=  15 / SIZE=    1 GRANS
SYS1/SYS SIP       LRL= 256 / EOF=   5 / SIZE=    1 GRANS
SYS2/SYS SIP       LRL= 256 / EOF=   5 / SIZE=    1 GRANS
SYS3/SYS SIP       LRL= 256 / EOF=   5 / SIZE=    1 GRANS
SYS4/SYS SIP       LRL= 256 / EOF=   5 / SIZE=    1 GRANS
SYS5/SYS SIP       LRL= 256 / EOF=   5 / SIZE=    3 GRANS
SYS6/SYS SIP       LRL= 256 / EOF=  15 / SIZE=    3 GRANS
FORMAT/CMD IP      LRL= 256 / EOF=  15 / SIZE=    3 GRANS
BACKUP/CMD IP      LRL= 256 / EOF=  15 / SIZE=    3 GRANS
TEST1/CMD  ↑       LRL= 256 / EOF=   6 / SIZE=    2 GRANS
           |             ↑            /           ↑
           |             |           /            |
DESIGNATORS          MOST        SIZE         SIZE IN
ARE "I" FOR        STANDARD       IN          GRANULES
INVISIBLE, "S"       FILES      SECTORS       (SEE DSHL)
FOR SYSTEM, "P"      HAVE
FOR PROTECTED      256-BYTE
                    RECORDS
```

DLM2
Directory Listing, Model II, TRSDOS

Read DIDI if you are not familiar with directory files.

To get a directory listing of all user files, do:

```
DIR :n
```

where :n is the disk drive number (:0-:4). The drive number is optional. If the drive number is omitted, drive 0 is assumed.

To get a directory listing of all user files and system files, do:

```
DIR :d (SYS)
```

To get a listing of the directory on the system line printer, use the PRT option, either with or without the SYS option:

```
DIR :1 (SYS,PRT)
```

The format of the listing is shown in Figure DLM2-1.

The first column of the listing is the file name and extension. A question mark following the file name indicates the file is still open or was improperly closed.

The second column is the creation date of the file.

The third column is the date of the last update.

The fourth column is a 4-character field defining the file attributes (see ADFC). The first character of this field is either P (program) or D (data). The second character is S (system) or * (user). The third character is the password status (X=unprotected, A=access password but no update, U=update password but no access, B=update and access passwords). The fourth character is the access level (0,1=kill, 2=rename, 3=unused, 4=write, 5=read, 6=execute, 7=none); see ADFC.

The fifth column is the record type, F for fixed length, and V for variable length.

The sixth column is the LRL, or logical record length.

The seventh column is the number of logical records in the file. Multiply the LRL by the # Rec to get the number of bytes in the file. A "+" sign here indicates no records have been written or the file has variable-length records.

The eighth column is the number of extents that are allocated to the file. An extent is a contiguous block of disk space.

The ninth column is the number of granules allocated for the file (see DSHL).

The tenth column is the number of sectors allocated for the file.

The file name is on the first line of the listing, and the last line has the number of free granules and extents.

Figure DLM2-1 – *Directory Listing, Model II TRSDOS*

```
                                                                      GRANULES ALLOCATED,
                                                                      SECTORS ALLOCATED,
                   DATA FILE,                            # OF RECORDS SECTORS USED
                   USER, UNPROTECTED                     IN FILE      (SEE DSHL)

DISK NAME:TRSDOS                     DRIVE:Ø             11/29/82       ØØ.Ø4.Ø1
 FILE NAME       CREATED    UPDATED   ATRB  FILE REC    NMBR      NMBR  GRAN SECT  SECT
               MM DD YY   MM DD YY        TYPE LEN    RECS      EXTS  ALOC ALOC  USED
TSTFLE         11 15 82   11 15 81   D*XØ   V  ...      ...        1    14   7Ø    66
ASM/SRC         1  1 81    1  1 82   D*XØ   V  ...      ...        1     2   1Ø     9
ASM             1  1 81    1  1 82   D*XØ   F  256        1        1     1    5     1
SORTPGM1/SRC    8 3Ø 82    8 3Ø 82   D*XØ   V  ...      ...        1     3   15    11
SORTPGM1        8 3Ø 82    8 3Ø 82   P*XØ   F  256        1        1     1    5     1
**  273 FREE GRANULES IN 2 EXTENTS **
                                   PROGRAM      LOGICAL RECORD  # OF CONTIGUOUS
                                   FILE, USER,      LENGTH         SEGMENTS
                                   UNPROTECTED RECORD TYPE
                                               V = VARIABLE LENGTH
                                               F = FIXED LENGTH
```

DLM3
Directory Listing, Model III, TRSDOS

Read DIDI if you are not familiar with directory files.

To get a directory listing of all visible (user) files, do:

```
DIR :n
```

where :n is the disk drive number (:0-:4). The drive number is optional. If the drive number is omitted, drive 0 is assumed.

To get a directory listing of all visible and invisible user files, do:

```
DIR :d (INV)
```

To get a directory listing of all visible and invisible user files and system files, do:

```
DIR :d (INV,SYS)
```

To get a listing of the directory on the system line printer, use the PRT option, and any of the other two options:

```
DIR :1 (INV,SYS,PRT)
```

The resulting listing is shown in Figure DLM3-1.

The first column of the listing is the file name and extension.

The second column is a 4-character field defining the file attributes (see ADFC). The first character of this field is either I (invisible) or N (visible). The second character is S (system) or U (user). The third character is the password status (X=unprotected, A=access password but no update, U=update password but no access, B=update and access passwords). The fourth character is the access level (0=total, 1=kill, 2=rename, 3=unused, 4=write, 5=read, 6=execute, 7=none); see ADFC.

The third column is the LRL, or logical record length.

The fourth column is the number of logical records in the file. Multiply the LRL by the # Rec to get the number of bytes in the file.

The fifth column is the number of granules allocated for the file (see DSHL).

The sixth column is the number of extents allocated to the file. An extent is a contiguous block of disk space.

The seventh column is the number of the last byte of the file.

The eighth column is the creation date of the file.

Figure DLM3-1 – *Directory Listing, Model III TRSDOS*

```
      INVISIBLE, SYSTEM FILE                                    CREATION
      UPDATE PASSWORD,                                            DATE
      ACCESS LEVEL = EXECUTE                                        |
 Disk Name: TRSDOS     |    Drive: Ø           Ø8/Ø5/81             ↓
 Filename        Attrb |    LRL     #Rec   #Grn   #Ext   EOF    Date
 BASIC/CMD        ISU6 ↓    256       2Ø      7      1     Ø    Ø5/81
 CONVERT/CMD      ISU6      256       1Ø      4      1     Ø    Ø5/81
 XFERSYS/CMD      ISU6      256        4      2      1     Ø    Ø5/81
 LPC/CMD          N*XØ      256        1      1      1     Ø    Ø5/81
 MEMTEST/CMD      N*XØ      256        8      3      1     Ø    Ø5/81
 HERZ5Ø/BLD       N*XØ      256        2      1      1    4Ø    Ø5/81
 TEST             N*XØ      256       46     16      2   171    Ø4/82
 PENCIL/CMD       N*XØ      256       2Ø      7      1     Ø    Ø4/82
 PENCILØ1/CMD     N*XØ      256       15      5      1     Ø    Ø4/82
 PENCILØ2/SYS     N*XØ      256        5      2      1     Ø    Ø4/82
 PENCILØ3/SYS     N*XØ      256       11      4      1     Ø    Ø4/82
 ***132 Free Granules***    ↑          ↑      ↑       \      \# OF LAST
             ↑             MOST    SIZE IN    |        \      FILE BYTE
 REMAINING                STANDARD  SECTORS  SIZE IN    \
 FREE SPACE               FILES HAVE         GRANULES   NUMBER OF
 (SEE DSHL)               256-BYTE           (SEE DSHL) CONTIGUOUS
                          RECORDS                       SEGMENTS
```

DLSC
Digital Logic, A Short Course

Confused by microprocessor chips, gates, and flip-flops? They're easy — they're only ones and zeroes compared to the black magic that goes on in your television set . . .

Signal levels: Digital circuits are either "on" or "off." If they are "ON" they are near +5 volts; this usually means from +3 volts to +4.95 volts. If the devices are off, they are near 0 volts, or ground. This usually means 0 volts to +0.5 volts.

A logic 1 is usually a high level or +5 volts. A logic 0 is usually a low level or 0 volts. Sometimes signals are "active low," which means they function when they go to zero volts instead of +5 volts. Many signals are "active high," which means they function when they go to +5 volts.

Purpose of digital logic: The purpose of digital logic is to string together high and low levels and have them define precise intervals at which things happen. If the right things happen at the same time and in the proper sequence, magic is evident, as in a functional digital computer.

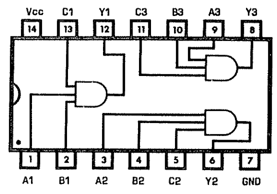
Timing of digital signals: You can look at digital signals generally as being on and off and going between these states instantaneously. Even though it does take some "rise" or "fall" time to swing from high to low or from low to high, it can be ignored in a properly designed circuit.

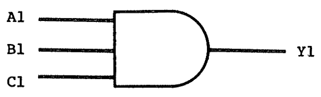
Types of digital circuits:

A. *AND gates*: AND gates take two or more signals and produce a 1 output when both inputs are high. Big deal, you've been through that in ANDs in BASIC, right? See Figure DLSC-1.

Figure DLSC-1A – *Simple Logic Devices*

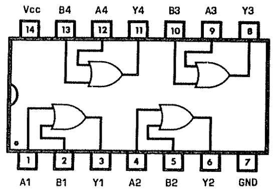
AND GATES

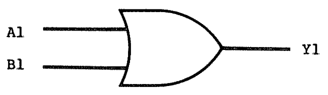




A1	B1	C1	Y1
Ø	Ø	Ø	Ø
Ø	Ø	1	Ø
Ø	1	Ø	Ø
Ø	1	1	Ø
1	Ø	Ø	Ø
1	Ø	1	Ø
1	1	Ø	Ø
1	1	1	1

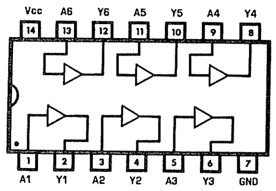
OR GATES

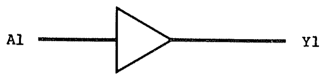




A1	B1	Y1
Ø	Ø	Ø
Ø	1	1
1	Ø	1
1	1	1

BUFFERS





A1	Y1
Ø	Ø
1	1

B. *OR gates*: OR gates take two or more signals and produce a 1 output when either one signal or another is high. See the figure.

C. *Buffers*: Buffers do not change the state, but simply "beef up" or isolate the electrical signal. A 0 is still a 0, a 1 still a 1.

D. *Inverters*: Inverters change a 1 to a 0 and a 0 to a 1.

E. *NAND Gates*: NAND gates are like AND gates, but output a 0 if all inputs are high (true).

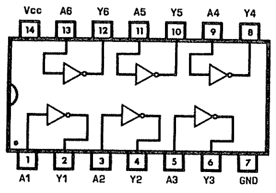
F. *NOR gates*: NOR gates are like OR gates, but output a 0 if an input is high (true).

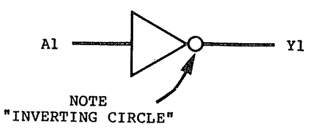
Okay, time to assimilate what you've learned . . . Suppose we have a combination lock that will activate a door latch every time the right combination is input. We can do it a number of different ways with the circuits shown in Figure DLSC-2. The combination code is 10101111.

So far we've talked about gates. These are devices that cannot remember anything. Let's discuss some memory devices:

Figure DLSC-1B – *Simple Logic Devices*

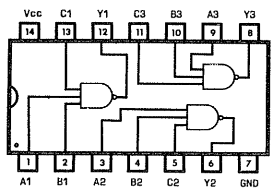
INVERTERS

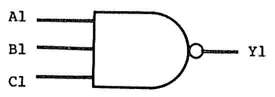




A1	Y1
Ø	1
1	Ø

NAND GATES

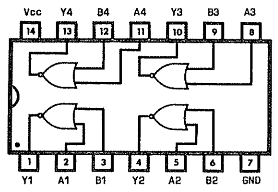


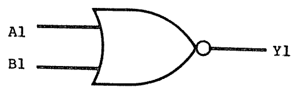


(COMPARE WITH AND)

A1	B1	C1	Y1
Ø	Ø	Ø	1
Ø	Ø	1	1
Ø	1	Ø	1
Ø	1	1	1
1	Ø	Ø	1
1	Ø	1	1
1	1	Ø	1
1	1	1	Ø

NOR GATES





A1	B1	Y1
Ø	Ø	1
Ø	1	Ø
1	Ø	Ø
1	1	Ø

G. *Flip-flops*: A flip-flop will store a single bit — a 0 or a 1. There are various types of flip-flops, but generally the input signal is "clocked" or "strobed in" on the "rising edge" or "falling edge" of some clock signal, as shown in Figure DLSC-3. Once the signal is clocked in, it will stay there until a new signal is clocked in or until a master CLEAR signal (0 or a 1, depending upon flip-flop) is clocked in or until a master CLEAR signal (0 or a 1) is given.

The purpose of a flip-flop is generally to remember data; a subordinate function is to record a high-speed pulse from microprocessor circuitry that occurs very rapidly. The flip-flop can match the high speed of the microprocessor to a slower-speed device.

H. *Registers*: What do you get when you assemble a batch of flip-flops? A register. A register is nothing more than 4 or 8 flip-flops arranged in one convenient package to remember 4 or 8 bits. Again, the bits are clocked in on the rising or falling edge of a clock signal.

I. *Counters*: Counters are special cases of registers. The input of each flip-flop in a counter is the output from the previous flip-flop. A 4-bit counter would count up from 0000, to 0001, to 0010, and so forth, on to 1111, at which point it would reset to 0000. Counters come in 4, 6, or 8 bits or more, count up or down, or in "decades" (tens), and generally have one output for each bit. They're used to define precise times by counting system clock pulses, or for counting any digital events.

J. *One-shots*: One shots are flip-flops with failing memory. When activated by a pulse, they stay at a 1 (or 0) for a predefined period of time from microseconds to milliseconds or longer, at which point they go low again until activated by another pulse. They're used for "time-outs" when the time does not have to be as accurately defined as by a counter.

Believe it or not, most of everything else is built upon these basic building blocks.

K. *Multiplexers*: These are essentially gates that pass 1 of 4 or 1 of 8 signals to an output based on the states of 2 or 3 "select" lines. See Figure DLSC-4.

L. *Decoders*: Again, these are gates that activate 1 of 4 or 8 outputs, based on the state of 2 or 3 select input signals. Widely used to decode "enable" signals based on memory or I/O address bits. See Figure DLSC-5.

See LDHT for a description of how to "read" computer logic diagrams, now that you know something about the components.

Figure DLSC-2 – *Sample Logic Design*

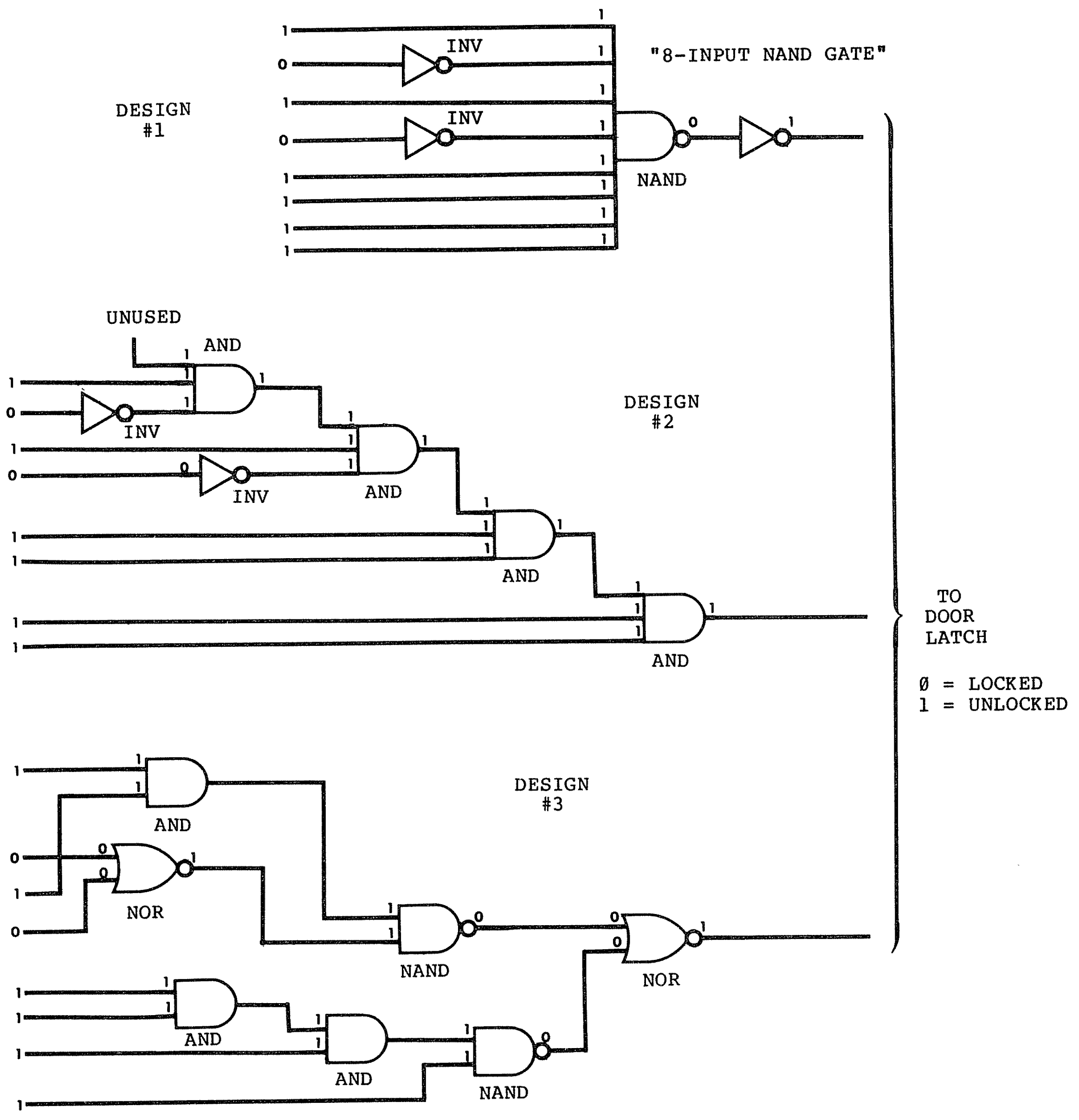
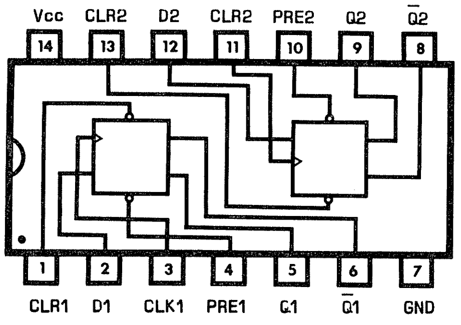
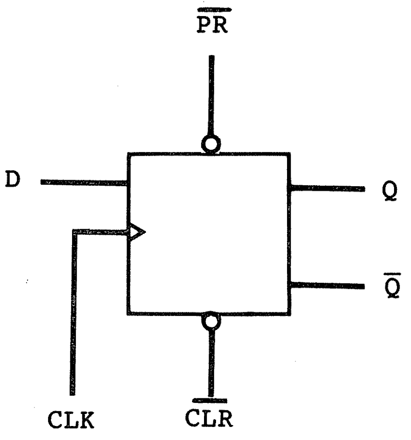


Figure DLSC-3 – *Flip-flop*





```
IF PR = Ø, THEN FLIP-FLOP IS "SET."
IF CLR = Ø, THEN FLIP-FLOP IS "RESET."
THESE INPUTS ARE USUALLY INFREQUENT.
```

CLK	D	Q	$\overline{Q}$
FROM Ø TO 1	Ø	Ø	1
FROM Ø TO 1	1	1	Ø
NO CHANGE	-	NO CHANGE	

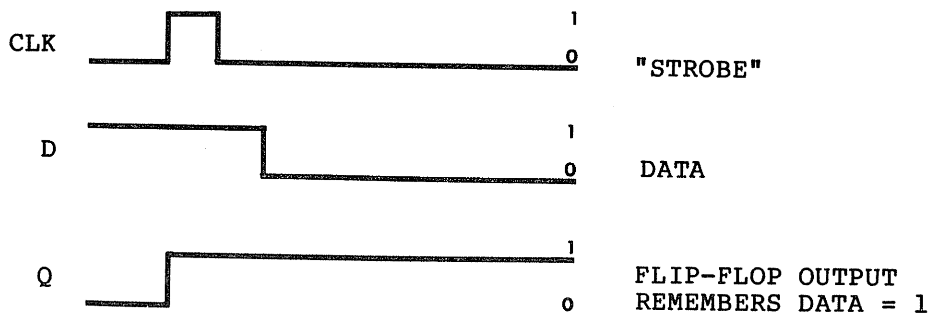
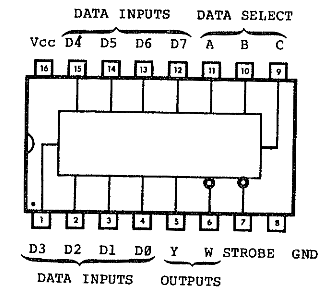


Figure DLSC-4 – *Multiplexer*

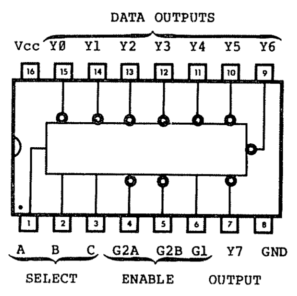


"STROBE" IS NORMALLY LOW (Ø)
"W" IS INVERSE OF "Y" OUTPUT

SELECT			OUTPUT
C	B	A	Y
Ø	Ø	Ø	DØ
Ø	Ø	1	D1
Ø	1	Ø	D2
Ø	1	1	D3
1	Ø	Ø	D4
1	Ø	1	D5
1	1	Ø	D6
1	1	1	D7

ONE OF 8 INPUTS SELECTED BASED ON SELECT SIGNALS

Figure DLSC-5 – *Decoder*



"ENABLE" SIGNALS ARE GENERAL SIGNALS TO PUT CHIP "ACTIVE"

SELECT			OUTPUT
A	B	C	
Ø	Ø	Ø	YØ = Ø, REST = 1
Ø	Ø	1	Y1 = Ø, " "
Ø	1	Ø	Y2 = Ø, " "
Ø	1	1	Y3 = Ø, " "
1	Ø	Ø	Y4 = Ø, " "
1	Ø	1	Y5 = Ø, " "
1	1	Ø	Y6 = Ø, " "
1	1	1	Y7 = Ø, " "

DLWS
Disk Light, When Should It Come On?

The red LED (light emitting diode) on the front of standard Radio Shack disk drives and many other drives should come on whenever data is being read from the disk or written to the disk. The disk drive motor on the I, III and Color Computer stays on a few seconds *after* the read or write operation, however, when really nothing is being done as far as transferring data. The II disk drive motor stays on continuously. Every time the red light on the disk of any system comes on, you know that data is being read or written to the disk.

DMT2
Dumping Memory to a Disk File, How to, Model II TRSDOS

The DUMP command in TRSDOS allows you to save a "memory image," This means that you can define a block of memory anywhere in the user area of memory and simply copy it to disk under a specified file name. Later, you can LOAD (see LMLD) the file from disk.

The block of memory can be either a machine-language program or data.

Use the DUMP command as follows:

```
DUMP name START=ssss,END=eeee
```

or

```
DUMP name START=ssss,END=eeee,TRA=tttt
```

The "name" parameter is a legitimate file name (see FNMH). The "ssss," "eeee" and "tttt" parameters are the start, end and transfer locations in memory in hexadecimal. Leading zeroes should not be used for hex addresses.

There is no default file extension. If you want to execute the dumped area as a machine language program, use /CMD.

The START address must be greater than **6FFFH**.

The TRAnsfer address is optional. Use it if you want to execute the dumped file as a /CMD program. If you haven't specified a transfer address, TRSDOS will use the start address.

See CFFT for file format of the dumped file.

To DUMP from one area and LOAD at another: use this form:

```
DUMP name START=ssss,END=eeee,TRA=tttt,RELO=rrrr
```

Example:

```
DUMP name START=8ØØØ,END=8FFF,TRA=AØØØ,RELO=AØØØ
```

This example will dump the area from **8000H** through **8FFFH**. A subsequent LOAD, though, will load the file into the area starting at **A000H**. The transfer address will be **A000H**.

The RELO option is useful for relocating some programs.

An optional parameter, RORT, can be used to designate that the DUMP file cannot be directly executed from TRSDOS. Use this form:

```
DUMP name
START=8ØØØ,END=8FFF,TRA=AØØØ,RELO=AØØØ,RORT=R
```

DMT3
Dumping Memory to a Disk File, How to, Model III TRSDOS

The DUMP command in TRSDOS allows you to save a "memory image." This means that you can define a block of memory anywhere in the user area of memory and simply copy it to disk under a specified file name. Later, you can LOAD (see LMLD) the file from disk.

The block of memory can be either a machine language program or data.

Use the DUMP command as follows:

```
DUMP name (START=ssss,END=eeee)
```

or

```
DUMP name (START=ssss,END=eeee,TRA=tttt)
```

The "name" parameter is a legitimate file name (see FNMH). The "ssss," "eeee" and "tttt" parameters are the start, end and transfer locations in memory in hexadecimal. Leading zeroes are required for hex addresses over **9FFFH**.

The default file extension is "/CMD". If you want to execute the dumped area as a machine language program, you won't need to RENAME (RADF).

The START address must be greater than **6FFFH**.

The TRAnsfer address is optional. Use it if you want to execute the dumped file as a /CMD program (see LMLD). If you haven't specified a transfer address, TRSDOS will use a default that goes back to the system.

See CFFT for file format of the dumped file.

To DUMP from one area and LOAD at another: use this form:

```
DUMP name (START=ssss,END=eeee,TRA=tttt,RELO=rrrr)
  EXAMPLE:
DUMP name (START=8ØØØ,END=8FFF,TRA=ØAØØØ,RELO=ØAØØØ)
```

This example will dump the area from **8000H** through **8FFFH**. A subsequent LOAD, though, will load the file into the area starting at **0A000H**. The transfer address will be **0A000H**.

The RELO option is useful for relocating machine language programs.

DMTL
Dumping Memory to a Disk File, How to, Model I/III LDOS

The DUMP command in LDOS allows you to save a "memory image." This means that you can define a block of memory anywhere in the user area of memory and simply copy it to disk under a specified file name. Later, you can LOAD (see LMLD) the file from disk. You can also create a pure ASCII file.

The block of memory can be either a machine language program or data.

Simple DUMPs: the simplest form of the DUMP command is:

```
DUMP name (START=X'ssss',END=X'eeee')
```

or

```
DUMP name (START=X'ssss',END=X'eeee',TRA=X'tttt')
```

The "name" parameter is a legitimate file name (see FNMH). The "ssss," "eeee" and "tttt" parameters are the start, end and transfer locations in memory in hexadecimal. Leading zeroes aren't required.

The default file extension is "/CIM". You might want to use "/CMD" if you want to execute the dumped area as a machine language program.

The START address must be greater than **5500H**.

The TRAnsfer address is optional. Use it if you want to execute the dumped file as a /CMD program (see LMLD). If not specified, LDOS will put in a default address back to the system.

See CFFT for file format of the dumped file.

Dumps in ASCII: use the ASCII option to dump an area of memory that contains ASCII data to create an ASCII file (see ADFW). This file will not contain any load information. The last character will be an "end of text" (ETX) character of X'03', unless you specify another character with the ETX option. Remember that the "end-of-file" is recorded in the directory, and that you don't strictly need the end-of-text; some programs, such as SCRIPSIT, however, look for a specific end-of-text marker.

To DUMP in ASCII:

```
DUMP name (START=X'ssss',END=X'eeee',TRA=X'tttt',ASCII)
```

or

```
DUMP name (START=X'ssss',END=X'eeee;TRA=X'tttt';
ASCII,EXT=X'FF')
```

The default extension is /TXT for ASCII dumps.

Other notes: Use S, E, T, A for abbreviations. Use the format S=ddddd for decimal addresses.

DMTT
Dumping Memory to a Disk File, How to, Model I TRSDOS

The DUMP command in TRSDOS allows you to save a "memory image." This means that you can define a block of memory anywhere in the user area of memory and simply copy it to disk under a specified file name. Later, you can LOAD (see LMLD) the file from disk.

The block of memory can be either a machine language program or data.

Use the DUMP command as follows:

```
DUMP name (START=X'ssss',END=X'eeee')
```

or

```
DUMP name (START=X'ssss',END=X'eeee',TRA=X'tttt')
```

The "name" parameter is a legitimate file name (see FNMH). The "ssss," "eeee" and "tttt" parameters are the start, end and transfer locations in memory in hexadecimal. Leading zeroes aren't required.

The default file extension is "/CIM". You might want to use "/CMD" if you want to execute the dumped area as a machine language program.

The START address must be greater than **6FFFH**.

The TRAnsfer address is optional. Use it if you want to execute the dumped file as a /CMD program (see LMLD). If you haven't specified a transfer address, TRSDOS will use a default that goes back to the system.

See CFFT for file format of the dumped file.

DPHU
Double-Precision Variables in BASIC, How to Use

(Does not apply to Model I/III, Level I or Color Computer).

Double-precision variables may be used to extend the accuracy of results. They use a similar floating-point representation as single-precision variables (SPHU), but extend the number of decimal digits to 17. Double-precision variables take up 8 bytes of storage and should be used only when great accuracy is required as storage and processing time is greater than other types of variables.

Specify a double-precision variable by the suffix "# ". In the following code, A # and XX # are double-precision variables:

```
100 A#=1234.44444456
110 XX#=234/.09888
```

A range of double-precision variables may also be specified by the DEFDBL command. DEFDBL A-G, for example, specifies all variables starting with an A through G as double-precision; AS, FD and GG would be double-precision in this case.

Using a suffix of "D" also denotes a double-precision number, in this case with scientific notation (see AOIB). The number must have been previously defined as a double-precision number by a # suffix or DEFDBL.

```
100 A#=123456.789D+5  'double precision
```

D

DMTL
DMTT

DPHU
DSBC

DSBC
Drawing Shapes in BASIC, Color Computer

See GHS1 for information on high-speed graphics using POKEs and strings. The same techniques can be used on the Color Computer in Color or Extended Color BASIC. See also GMIC.

LINE: LINE is somewhat of a misnomer — it should be called, in a rush of breath, "LINEBOXFILLEDINBOX." (Microsoft originally wanted to use this as the command, but cooler heads prevailed).

LINE will draw a line between any two points, as in

```
100 LINE (X1,Y1)-(X2,Y2),PSET
```

which draws the line in the foreground color.

The average line is drawn in about 96 milliseconds, and the worst case is about 192 milliseconds, about 20 times faster than the fastest BASIC line drawing routine.

LINE will also draw a box (rectangle) outline. In this case the two coordinates specify the opposing corners of the box.

```
100 (50,50)-(60,60),PSET,B    'B specifies box
```

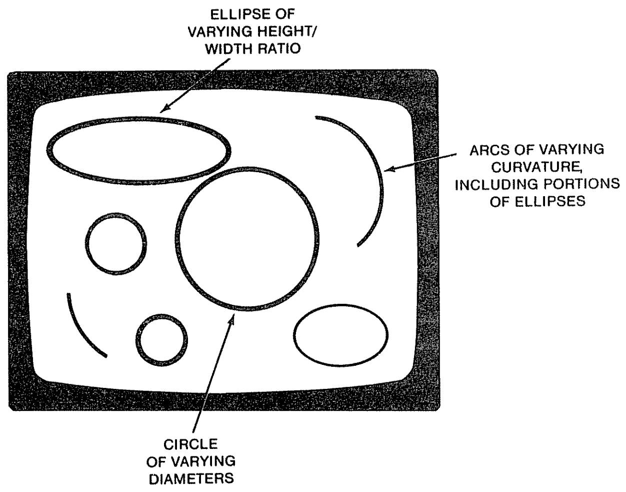
A third use is in drawing a "filled-in" box. The box is drawn at speeds comparable to drawing four lines. The filled in box (foreground color is used) is, of course, a lot slower (the time may go over one second for large boxes), but still excellent for such a powerful command:

```
100 (50,50)-(60,60),pset,BF    'BF is filled-in box
```

CIRCLE: Again, CIRCLE was originally named CIRCLEELLIPSEARC, as it draws circles, ellipses, and arcs of circles or ellipses, as shown in Figure DSBC-1.

When CIRCLE is used for any of the three types of figures, the center must be within screen boundaries; this prevents *all* arcs from being drawn; an arc close to the edge of the screen, for example, is not possible, as the center of the circle or ellipse on which it lies is outside of screen boundaries, as shown in Figure DSBC-2.

Figure DSBC-1 – *CIRCLE Action*


ELLIPSE OF VARYING HEIGHT/ WIDTH RATIO
ARCS OF VARYING CURVATURE, INCLUDING PORTIONS OF ELLIPSES
CIRCLE OF VARYING DIAMETERS

```
100 CIRCLE (50,50),10
```

draws a circle of radius 10 at 50,50, while

```
110 CIRCLE (50,50),10,c
```

specifies a color c (see GMIC), while

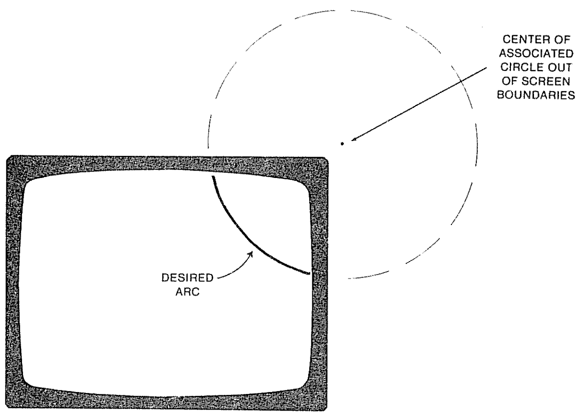
```
120 CIRCLE (50,50),10,c,hw
```

draws an ellipse, with hw defining the height/width ratio (0 through large values), while

```
13Ø CIRCLE (5Ø,5Ø),1Ø,c,hw,st,en
```

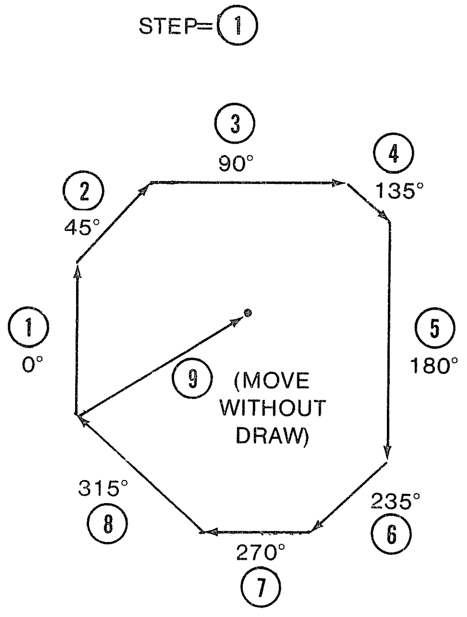
draws a circle or ellipse from a start (st) to end (en) value. Start and end must be 0 through 1 and define the starting and ending points on an arc; three o'clock is 0 and going clockwise back to three o'clock is 1.

Figure DSBC-2 – *Arc Which Cannot be Drawn*



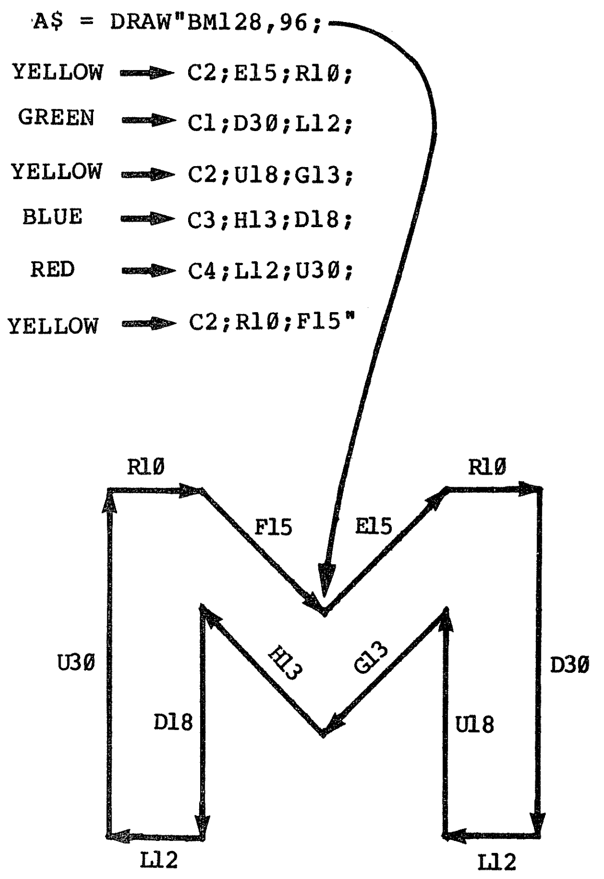
DRAW draws a series of line segments in multiplies of 45 degrees, as shown in Figure DSBC-3. The line segments may be of any length. In addition to drawing line segments, DRAW will position the cursor to a specific spot on the screen, change the color of the line segment, rotate a figure in 90 degree increments, execute a substring, and change the scale of the lines to be drawn.

Figure DSBC-3 – *DRAW Line Segment Action*



Suppose that we want to draw the letter "M." We can easily draw it by the code in Figure DSBC-4. In the code, the color is changed for different line segments with the C subcommand.

Figure DSBC-4 – *Use of the DRAW Command*



```
A$ = DRAW"BM128,96;
YELLOW  ➡ C2;E15;R1Ø;
GREEN   ➡ C1;D3Ø;L12;
YELLOW  ➡ C2;U18;G13;
BLUE    ➡ C3;H13;D18;
RED     ➡ C4;L12;U3Ø;
YELLOW  ➡ C2;R1Ø;F15"
```

To rotate the M through 90 degrees, we'd simply add an "A" (angle) subcommand as part of the string before the DRAW string, as shown in Figure DSBC-5.

To change the size of the "M," we'd add an "S" (scale) subcommand before the DRAW string, as shown in Figure DSBC-6. The scale factor can be changed from 1/4 to 62/4 of the original size of the figure. You can see how this could be a great advantage in generating all types of figures that change size.

Probably the most powerful feature of DRAW is the ability to execute "substrings." We could define figure 1 as string A$, figure 2 as string B$, and figure 3 as string C$. A fourth string could then "execute" (by the X subcommand) the other strings to build up composite figures, as shown in Figure DSBC-7.

DRAW can be used for many applications. One that comes immediately to mind is defining different character sets for the Color Computer. There are 256 pixels across the screen and 192 down, and you can see how characters representing Greek, Kata-Kana or others could be defined by working with matrices of 8 by 12 pixels (32 characters by 16 lines) or larger matrices.

Figure DSBC-5 – *Rotation of Figures Using DRAW*

```
100    A$="BM128,96;. . . ."
110    DRAW "A0"+A$
120    DRAW "A1"+A$
130    DRAW "A2"+A$
140    DRAW "A3"+ A$
```

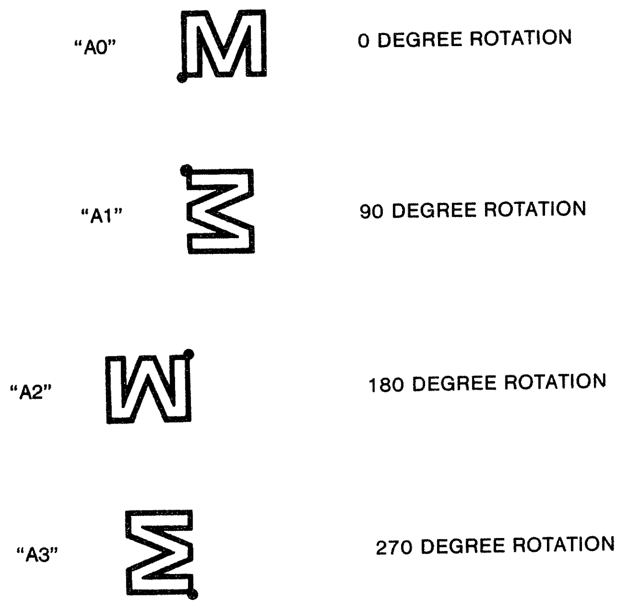
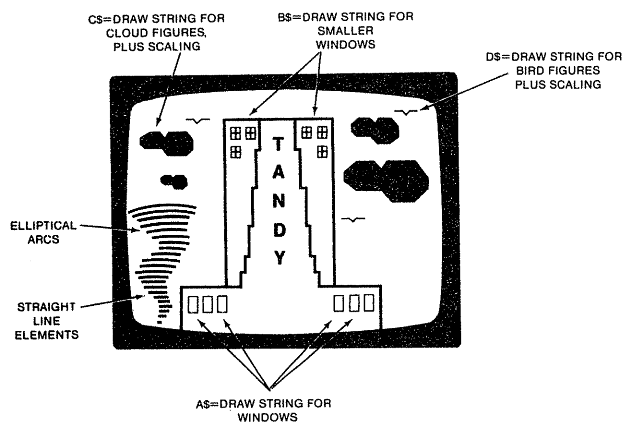


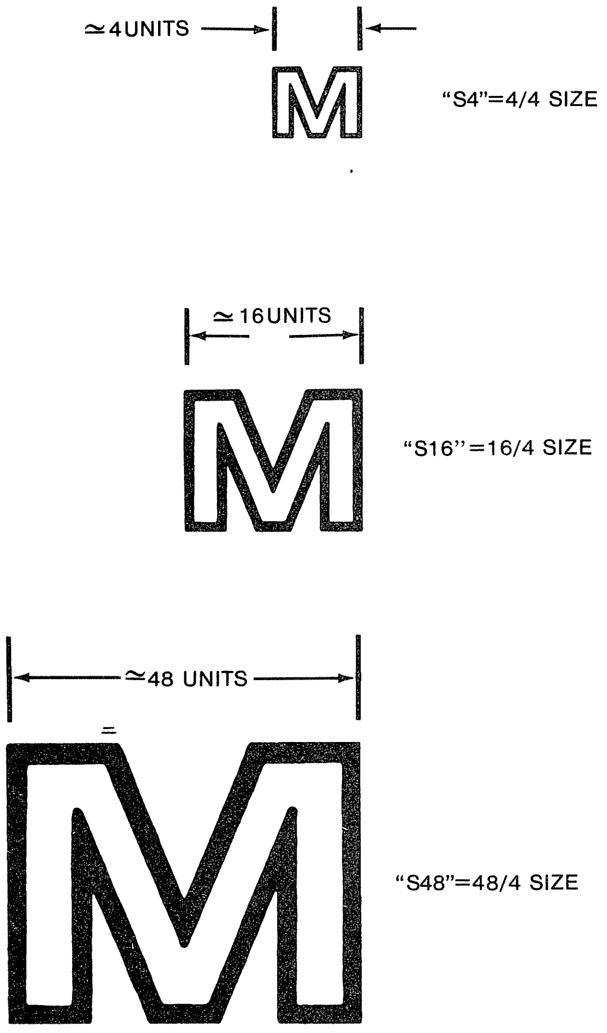
Figure DSBC-7 – *Substring Use in DRAW*



C$=DRAW STRING FOR CLOUD FIGURES, PLUS SCALING
B$=DRAW STRING FOR SMALLER WINDOWS
D$=DRAW STRING FOR BIRD FIGURES PLUS SCALING
T
A
N
D
Y
ELLIPTICAL ARCS
STRAIGHT LINE ELEMENTS
A$=DRAW STRING FOR WINDOWS

Figure DSBC-6 – *Scaling of Figure with the DRAW command*

```
100    A$="BM128, 96;. . . ."
110    DRAW "S4"+A$
120    DRAW "S16"+A$
130    DRAW "S48"+A$
```



D

DSBC

DSBC

DSDS
Date, Setting, Model I/II/III TRSDOS/LDOS

To reset the date on the Models I and III (it is mandatory on TRSDOS/LDOS load), enter the current date in this format:

```
DATE MM/DD/YY
```

where MM, DD, and YY are the month, day, and year, each in two digits (use a leading 0 if necessary, as in 01/02/82).

Model II users: enter

```
DATE MM/DD/YYYY
```

where YYYY is the 4-digit year.

Depending upon the system, the date will be updated at midnight, assuming the TIME has been properly set (see TSRT). However, none of the Radio Shack systems should be used as accurate clocks (see RTCN), even though the software designers did include "real-time-calenders." DATE is used primarily to update directory file entries, and although it's a nuisance to enter each time you power up the system, it is valuable to have dated directory entries.

Enter DATE without any arguments to find out the current date (see DFOT).

DSHL
Disk Space, How Much Left? TRSDOS, All Systems

On the Model I, enter Free. TRSDOS will reply with something like

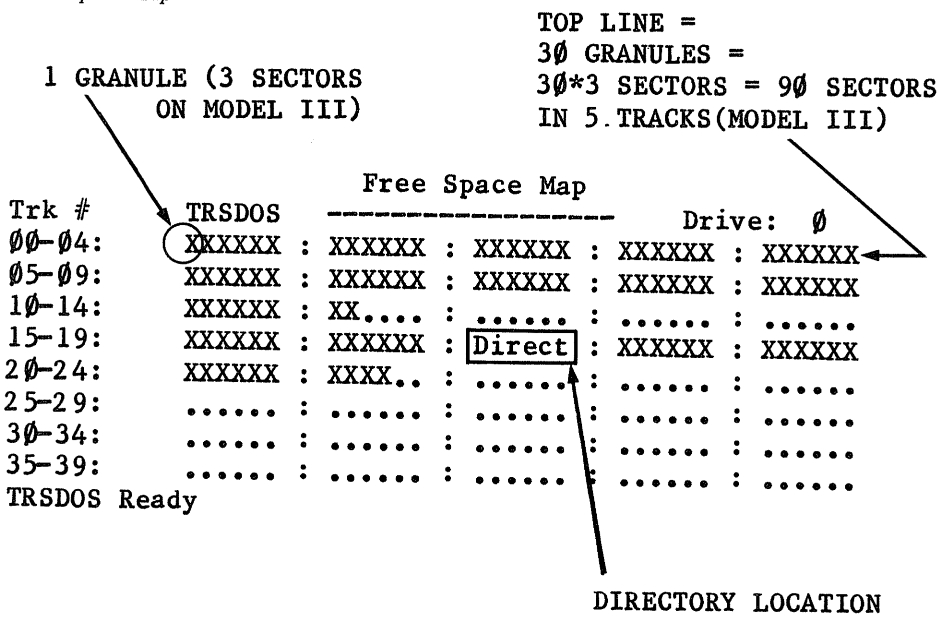
```
DRIVE Ø -- TRSDOS  11/11/82   23 FILES,   3Ø GRANS
DRIVE 1 -- TRSDOS  12/12/82   17 FILES,   4Ø GRANS
```

See below for GRAN conversion.

On the Model II or III, enter FREE, followed by colon, followed by drive number (example: FREE :2) or simply FREE for drive 0 after the TRSDOS READY prompt, or FREE :n(PRT) for listing on system line printer (use FREE :n PRT in Model II).

TRSDOS will display or print a "map" of the disk space, as shown in Figure DSHL-1. In this map, a period represents an unused "granule", a X is an allocated granule, "Direct" is the location of the diskette directory, and "Flawed" marks a flawed, or unusable sector.

Figure DSHL-1 – *Disk Space Map*



```
                        Free Space Map
Trk #      TRSDOS    ----------------      Drive:  Ø
ØØ-Ø4:     XXXXXX  :  XXXXXX  :  XXXXXX  :  XXXXXX  :  XXXXXX
Ø5-Ø9:     XXXXXX  :  XXXXXX  :  XXXXXX  :  XXXXXX  :  XXXXXX
1Ø-14:     XXXXXX  :  XX....  :  ......  :  ......  :  ......
15-19:     XXXXXX  :  XXXXXX  :  Direct  :  XXXXXX  :  XXXXXX
2Ø-24:     XXXXXX  :  XXXX..  :  ......  :  ......  :  ......
25-29:     ......  :  ......  :  ......  :  ......  :  ......
3Ø-34:     ......  :  ......  :  ......  :  ......  :  ......
35-39:     ......  :  ......  :  ......  :  ......  :  ......
TRSDOS Ready
```

Color Computer: Enter

```
PRINT FREE(n)
```

where n is the drive number. The number of free granules for the diskette in the drive will be printed.

Granules: A granule is simply an arbitrary number of sectors used in allocating disk files. The smallest workable division on a disk is a sector, due to the way data is stored on a disk. It's reasonable, then, to allocate space on a sector basis, or a sector "multiple" basis, and that's what's done in dividing a disk into granules, or sector multiples. One granule on the Model I is 5 sectors, on the Model II is 5 sectors, on the Model III is 3 sectors, and on the Color Computer is 3 sectors. A sector is 256 bytes. To find the number of free bytes, multiply the number of free granules by the number of sectors per granule, and that result by the number of bytes per sector.

System	Tracks/ Diskette	Sec- tors/ Trk	Bytes/ Sector*	Sec- tors/ Gran	Bytes/ Gran	Max Disk Bytes	Max Disk Grans
I	35	1Ø	256	5	128Ø	89,6ØØ	7Ø
II	77	26	256	5	128Ø	5Ø9,184*	395
III	4Ø	18	256	3	768	184,32Ø	24Ø
CC	35	18	256	3	768	161,28Ø	21Ø

*Track Ø has 128 bytes/sector

DSPR
Dumping the Screen to the Printer in BASIC, Model I/III

Model III only:

Method 1: at any time in BASIC (except when using cassette, printer or serial input/output), press SHIFT, down arrow and asterisk at the same time. This prints the screen contents onto the system line printer.

Method 2: during BASIC program execution, call location 473 in ROM by a USR call (see CML1 or CMD1); no arguments are necessary. Sample for Model III, non-disk BASIC:

```
1ØØ POKE 16526,217     'ls byte of address
11Ø POKE 16527,1       'ms byte of address
12Ø A=USR(Ø)           'print screen contents
13Ø ...                'back here
```

Sample for TRSDOS Disk BASIC:

```
1ØØ DEFUSRØ=473        'define address
11Ø A=USRØ(Ø)          'print screen contents
12Ø ...                'back here
```

The format output to the line printer will be 64-character lines, 16 lines total, to reproduce the entire screen. Non-displaying or graphics characters will be printed as periods.

Method 3 (Disk BASIC but not LDOS LBASIC): execute

```
CMD"Z","ON"
```

This will duplicate output to the screen on the line printer. The current screen will not be printed, but all screen lines after the CMD"Z" will also be sent to the printer. (See "DUAL" in PTSC.) Turn off the screen routing by

```
CMD"Z","OFF"
```

Model I/III LDOS:

Execute CMD"*" while in LBASIC. This operates similarly to method 1, above.

See also "LINK" under PTSC.

DSRC
DATA Statements and Related Commands, BASIC, All Systems

DATA statements are used to establish tables of data, either numeric or string. The lines

```
1ØØ DATA 1ØØ,4.5,APPLE
11Ø DATA 2ØØ,9.3,PEAR
```

establish a table of six data items, arranged in the sequence that they appear in the DATA statements.

Numeric data and string data can be intermixed at will. String data can be enclosed by double quotes as in other strings (see SHTU), or can be without the double quotes as shown in the examples. If the strings have leading blanks or embedded commas or colons, then the double quotes are mandatory.

You can access the data in the DATA statements by a READ command. The READ command "reads" the data one item at a time in the order in which it appears in the DATA statements.

```
5ØØ READ A,B,A$,C,D,B$
```

for example, reads the data items above into numeric variables A, B, string variable A$, numeric variables C, D and string variable B$. You must READ the data in the same order and with the same data "type" (numeric or string) as the data is ordered in the DATA statements.

Look on all DATA statements as forming one huge table. DATA statements don't have to be "contiguous," that is, you can have two DATA statements, followed by other BASIC lines, followed by two more DATA statements, and so forth. You also don't have to place the

DATA statements in any special area of the program or "jump" around them; you can place them right in the middle of BASIC code.

When you start reading the data from DATA statements, subsequent READs will read the next entry from the DATA "table" until the last entry is read. Naturally, you have to have the same number of READs as there are data values, otherwise you'll get an "Out of Data" error (see ODER).

To reset the imaginary "pointer" to the next item in the DATA table, execute a RESTORE command. The RESTORE "resets" the DATA table pointer to the beginning of the DATA list:

```
100 DATA 2,3,7,67,8.9
110 READ A,B,C              'reads 2, 3, 7
120 RESTORE                 'reset
130 READ D                  'reads 2
```

DT1U
DEBUG, TRSDOS/LDOS, Model I/III, Using

D

DEBUG is a "monitor" program to:

Allow you to examine the contents of all memory

Examine and change the contents of Disk Files

Help you debug machine language programs

DEBUG is loaded from TRSDOS

```
TRSDOS READY
DEBUG
```

On the Model III, this loads and executes DEBUG; on the Model I, press BREAK after the DEBUG entry to execute.

Model I and III DEBUG are almost identical, but there are slight differences. I'll point them out as we go along.

1. **To examine memory**:

   A. Load and execute DEBUG.

   B. You'll see a display of memory and Z-80 cpu registers as shown in Figure DT1U-1 (Model I) or DT1U-2 (Model III).

   C. On TRSDOS Model I and LDOS DEBUG, enter DXXXX, where XXXX is the hexadecimal starting address of the memory you want displayed (see MMM1 for memory map and CFDH for hexadecimal).

   On the Model III TRSDOS DEBUG, enter D; DEBUG will respond with "D ADDRESS="; enter the hexadecimal address of the memory area you want displayed, XXXX.

   D. DEBUG will now display the memory area in the same format as in Figure DT1U-1 or DT1U-2.

2. **To get a full screen display**: The display above is "half-screen." Memory and cpu registers are displayed. To get a full-screen display of memory only:

   A. Type S.

3. **To get back to half-screen display**:

   A. Type X to display registers and memory.

4. **To "scroll" forward and back** to display previous or following memory locations:

   A. Press semicolon (;) to "scroll" ahead to the next half-screen or full screen worth of memory data.

   B. Press minus (-) to "scroll" back to the last half-screen or full screen worth of memory data.

   C. You can continue pressing the semicolon or minus key to get a continuous display of memory until you find the area you are looking for.

5. **To change the contents of a memory location**:

   A. Before changing the contents, make certain you know what you are changing! It's much easier to clobber the system with DEBUG than with BASIC!

   B. On TRSDOS Model I and LDOS DEBUG, type MXXXX (space bar), where XXXX is the hexadecimal value of the location to modify.

   On TRSDOS Model III DEBUG, type M, and reply to the "M ADDRESS" with the value of the location to modify, XXXX.

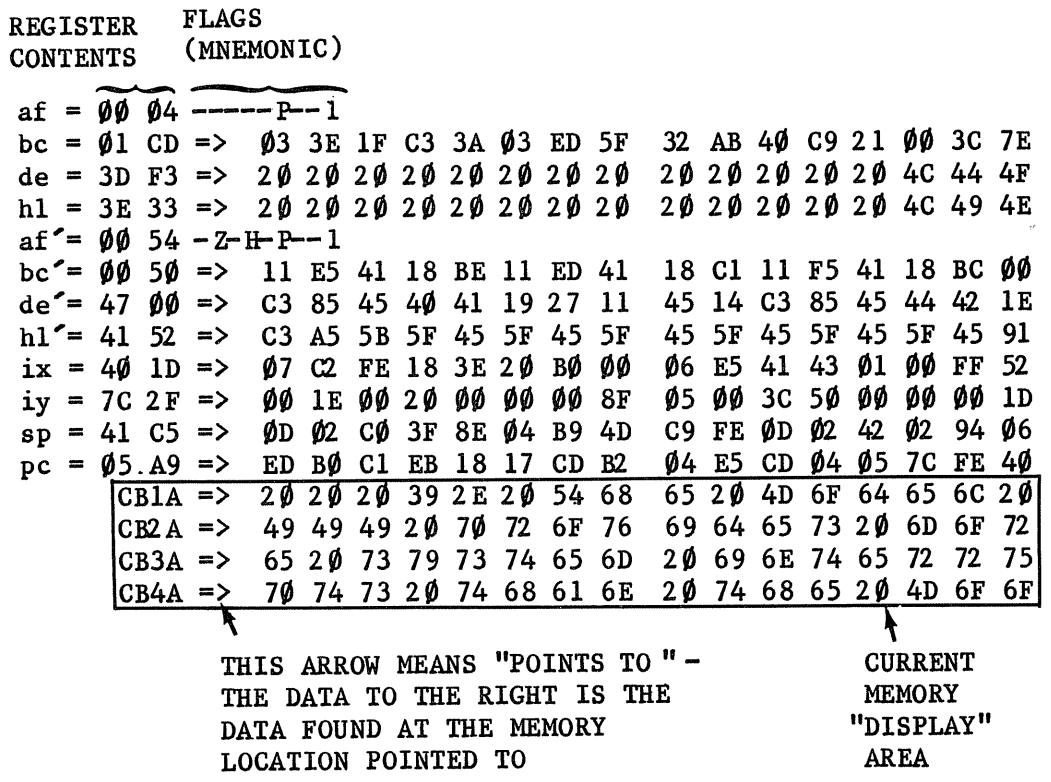
   Note that DEBUG expects 4 hexadecimal digits (see CFDH) and will not do anything until those 4 digits are typed.

   C. DEBUG responds with the contents of the memory location, as indicated in Figure DT1U-3 or DT1U-4. Note also that if the location is being displayed in the memory area (it may or may not be, depending upon whether you are displaying the memory area which you're about to modify), vertical bars surround the location of the data.

   D. Up to this point, nothing has been changed. To go to the next location without changing anything, type SPACE BAR. To change the current data, type in 1 to 4 hexadecimal digits for the change and type SPACE BAR. To get back to the DEBUG command mode (and stop the Modify), press ENTER.

E. You can change consecutive data rapidly by typing in new data, interspersed with blanks for "no change."

Figure DT1U-1 – *Model I DEBUG Display*



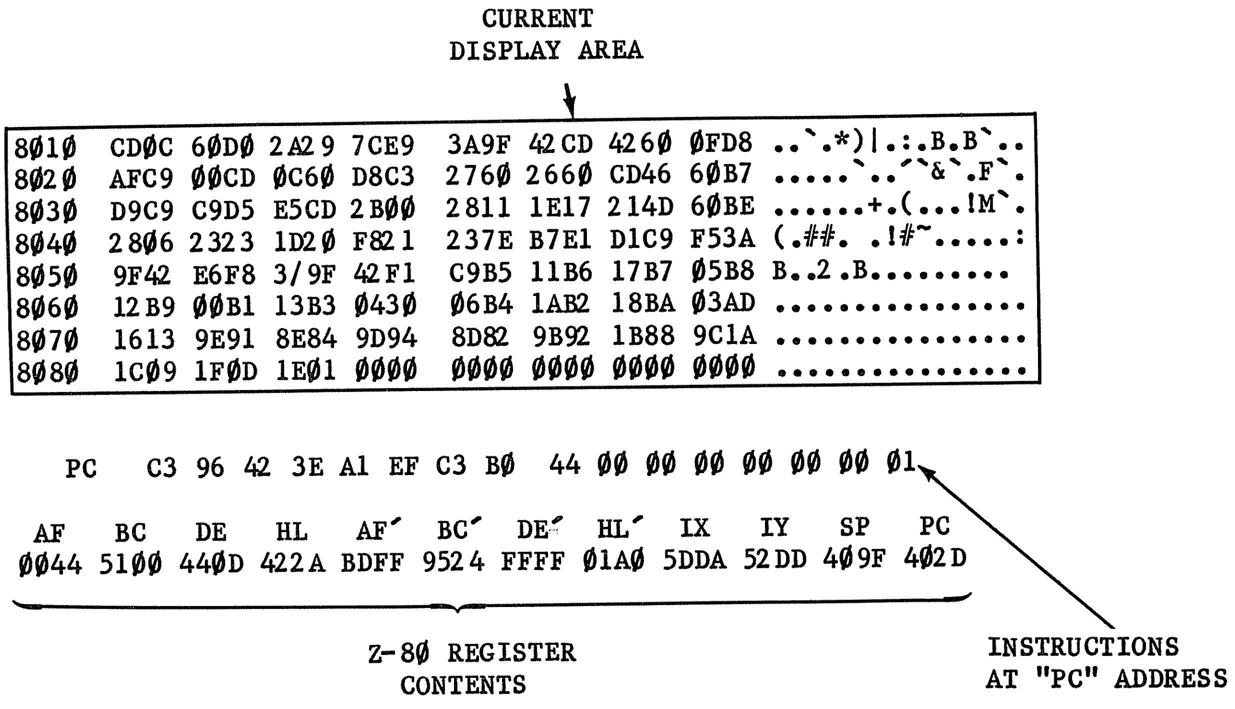
```
REGISTER   FLAGS
CONTENTS   (MNEMONIC)

af = ØØ Ø4 ------P--1
bc = Ø1 CD =>  Ø3 3E 1F C3 3A Ø3 ED 5F   32 AB 4Ø C9 21 ØØ 3C 7E
de = 3D F3 =>  2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø   2Ø 2Ø 2Ø 2Ø 2Ø 4C 44 4F
hl = 3E 33 =>  2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø   2Ø 2Ø 2Ø 2Ø 2Ø 4C 49 4E
af'= ØØ 54 -Z-H-P--1
bc'= ØØ 5Ø =>  11 E5 41 18 BE 11 ED 41   18 C1 11 F5 41 18 BC ØØ
de'= 47 ØØ =>  C3 85 45 4Ø 41 19 27 11   45 14 C3 85 45 44 42 1E
hl'= 41 52 =>  C3 A5 5B 5F 45 5F 45 5F   45 5F 45 5F 45 5F 45 91
ix = 4Ø 1D =>  Ø7 C2 FE 18 3E 2Ø BØ ØØ   Ø6 E5 41 43 Ø1 ØØ FF 52
iy = 7C 2F =>  ØØ 1E ØØ 2Ø ØØ ØØ ØØ 8F   Ø5 ØØ 3C 5Ø ØØ ØØ ØØ 1D
sp = 41 C5 =>  ØD Ø2 CØ 3F 8E Ø4 B9 4D   C9 FE ØD Ø2 42 Ø2 94 Ø6
pc = Ø5.A9 =>  ED BØ C1 EB 18 17 CD B2   Ø4 E5 CD Ø4 Ø5 7C FE 4Ø
     CB1A  =>  2Ø 2Ø 2Ø 39 2E 2Ø 54 68   65 2Ø 4D 6F 64 65 6C 2Ø
     CB2A  =>  49 49 49 2Ø 7Ø 72 6F 76   69 64 65 73 2Ø 6D 6F 72
     CB3A  =>  65 2Ø 73 79 73 74 65 6D   2Ø 69 6E 74 65 72 72 75
     CB4A  =>  7Ø 74 73 2Ø 74 68 61 6E   2Ø 74 68 65 2Ø 4D 6F 6F
```

THIS ARROW MEANS "POINTS TO" – THE DATA TO THE RIGHT IS THE DATA FOUND AT THE MEMORY LOCATION POINTED TO

CURRENT MEMORY "DISPLAY" AREA

Figure DT1U-2 – *Model III DEBUG Display*



CURRENT DISPLAY AREA

```
8Ø1Ø  CDØC 6ØDØ 2A29 7CE9  3A9F 42CD 426Ø ØFD8 ..`.*)|.:.B.B`..
8Ø2Ø  AFC9 ØØCD ØC6Ø D8C3  276Ø 266Ø CD46 6ØB7 .....`..'`&`.F`.
8Ø3Ø  D9C9 C9D5 E5CD 2BØØ  2811 1E17 214D 6ØBE ......+.(...!M`.
8Ø4Ø  28Ø6 2323 1D2Ø F821  237E B7E1 D1C9 F53A (.##. .!#~.....:
8Ø5Ø  9F42 E6F8 3/9F 42F1  C9B5 11B6 17B7 Ø5B8 B..2.B.........
8Ø6Ø  12B9 ØØB1 13B3 Ø43Ø  Ø6B4 1AB2 18BA Ø3AD ................
8Ø7Ø  1613 9E91 8E84 9D94  8D82 9B92 1B88 9C1A ................
8Ø8Ø  1CØ9 1FØD 1EØ1 ØØØØ  ØØØØ ØØØØ ØØØØ ØØØØ ................

PC   C3 96 42 3E A1 EF C3 BØ  44 ØØ ØØ ØØ ØØ ØØ ØØ Ø1

AF   BC   DE   HL   AF'  BC'  DE'  HL'  IX   IY   SP   PC
ØØ44 51ØØ 44ØD 422A BDFF 9524 FFFF Ø1AØ 5DDA 52DD 4Ø9F 4Ø2D
```

Z-8Ø REGISTER CONTENTS

INSTRUCTIONS AT "PC" ADDRESS

6. **To change the contents of a CPU register**:

A. This is normally done prior to transferring control to a machine language program, otherwise it has no effect.

B. TRSDOS Model I and LDOS DEBUG: Type RYYbXXXX SPACE BAR, where YY is a register pair "mnemonic" (see below), b is a blank (space bar), and XXXX is 1- to 4-digit hexadecimal value (see CFDH).

TRSDOS Model III DEBUG: Type RYY,XXXX (space bar), where YY is a register pair "mnemonic" (see below) and XXXX is a 4-digit hexadecimal value.

Register Pair Mnemonics:

AF
BC
DE
HL
IX
IY
SP
PC

C. You should see the register pair contents change if you are in the half-screen mode. By the way, the XXXX value is not in "reverse notation" as is normally done in POKEs to two locations representing 16 bits of data. If the register specified is BC and the value entered is **1A3F**, **1AH** will be put into the B register, and **3FH** will be put into the C register.

D. Repeat these steps to change other register pairs, as required.

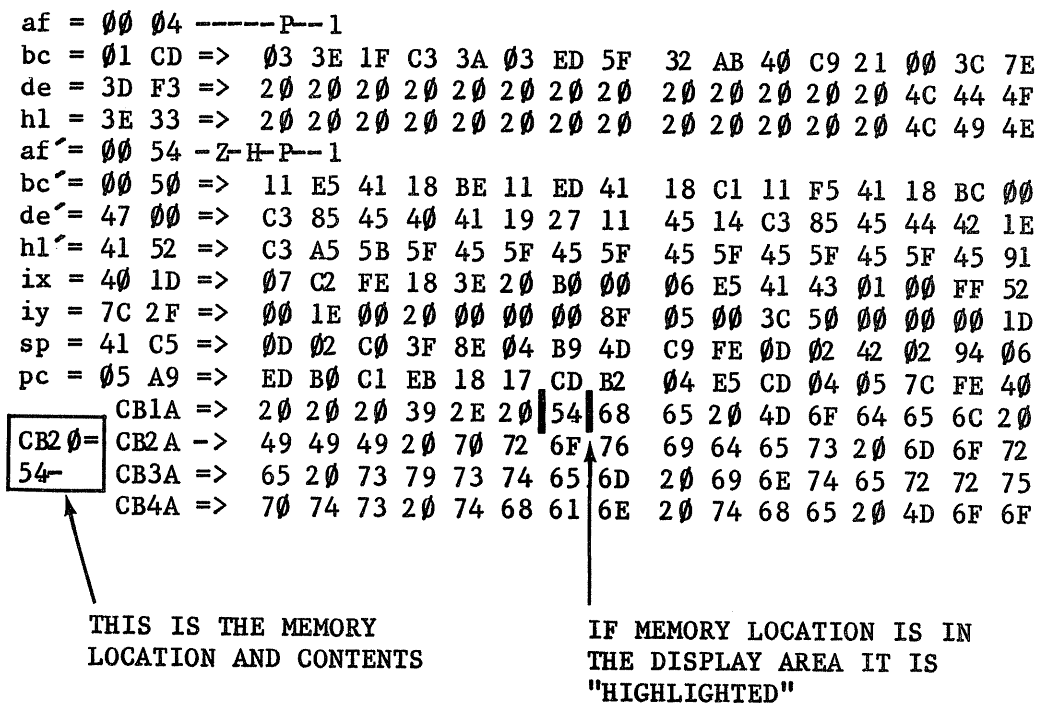
7. **To transfer control to a machine language program**:

A. Change the register contents as required by step 6 above. Most machine-language code requires the CPU registers to be set up to certain values.

B. In TRSDOS Model I and LDOS DEBUG: Type GXXXX ENTER to transfer control to location XXXX. This will cause a Jump to location XXXX, without "breakpoints" (see BPFM). If you want breakpoints, type in as many breakpoints as you require (except 2 for LDOS), with a comma between the transfer address and each breakpoint:

```
G8ØØØ,8Ø1Ø,8Ø2Ø ENTER
```

Figure DT1U-3 – *Model I 'M' Action*



```
af = ØØ Ø4 ------P--1
bc = Ø1 CD =>  Ø3 3E 1F C3 3A Ø3 ED 5F   32 AB 4Ø C9 21 ØØ 3C 7E
de = 3D F3 =>  2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø   2Ø 2Ø 2Ø 2Ø 2Ø 4C 44 4F
hl = 3E 33 =>  2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø   2Ø 2Ø 2Ø 2Ø 2Ø 4C 49 4E
af'= ØØ 54 -Z-H-P--1
bc'= ØØ 5Ø =>  11 E5 41 18 BE 11 ED 41   18 C1 11 F5 41 18 BC ØØ
de'= 47 ØØ =>  C3 85 45 4Ø 41 19 27 11   45 14 C3 85 45 44 42 1E
hl'= 41 52 =>  C3 A5 5B 5F 45 5F 45 5F   45 5F 45 5F 45 5F 45 91
ix = 4Ø 1D =>  Ø7 C2 FE 18 3E 2Ø BØ ØØ   Ø6 E5 41 43 Ø1 ØØ FF 52
iy = 7C 2F =>  ØØ 1E ØØ 2Ø ØØ ØØ ØØ 8F   Ø5 ØØ 3C 5Ø ØØ ØØ ØØ 1D
sp = 41 C5 =>  ØD Ø2 CØ 3F 8E Ø4 B9 4D   C9 FE ØD Ø2 42 Ø2 94 Ø6
pc = Ø5 A9 =>  ED BØ C1 EB 18 17 CD B2   Ø4 E5 CD Ø4 Ø5 7C FE 4Ø
        CB1A => 2Ø 2Ø 2Ø 39 2E 2Ø 54 68   65 2Ø 4D 6F 64 65 6C 2Ø
CB2Ø=   CB2A -> 49 49 49 2Ø 7Ø 72 6F 76   69 64 65 73 2Ø 6D 6F 72
54-     CB3A => 65 2Ø 73 79 73 74 65 6D   2Ø 69 6E 74 65 72 72 75
        CB4A => 7Ø 74 73 2Ø 74 68 61 6E   2Ø 74 68 65 2Ø 4D 6F 6F
```

THIS IS THE MEMORY LOCATION AND CONTENTS
IF MEMORY LOCATION IS IN THE DISPLAY AREA IT IS "HIGHLIGHTED"

Figure DT1U-4 – *Model III 'M' Action*

```
8Ø1Ø: CDØC 6ØDØ 2A29 7CE9 3A9F 42CD 426Ø ØFD8 ..`.*)|.:.B.B`..
8Ø2Ø: AFC9 ØØCD ØC6Ø D8C3 276Ø 266Ø CD46 6ØB7 .....`..´`&`.F`.
8Ø3Ø: D9C9 C9D5 E5CD 2BØØ 2811 1E17 214D 6ØBE ......+.(...!M`.
8Ø4Ø: 28Ø6 2323 1D2Ø F821 237E B7E1 D1C9 F53A (.##. .!#~.....:
8Ø5Ø: 9F42 E6F8 3/9F 42F1 C9B5 11B6 17B7 Ø5B8 B..2.B.........
8Ø6Ø: 12B9 ØØB1 13B3 Ø43Ø Ø6B4 1AB2 18BA Ø3AD ................
8Ø7Ø: 1613 9E91 8E84 9D94 8D82 9B92 1B88 9C1A ................
8Ø8Ø: 1CØ9 1FØD 1EØ1 ØØØØ ØØØØ ØØØØ ØØØØ ØØØØ ................
8Ø9Ø: ØØFE 2ACA 71█  FE23 2821 FE24 C276 75CD ....qu.#(!.$.vu.
8ØAØ: DØ6Ø 28EF 13D5 EBB7 CD4C 6C7D CDØ2 52D1 .`(......L1}..R.
8ØBØ: CD97 76CD FE53 2ØEC 13B7 C9CD DØ6Ø 28D3 ..v..S ......`(.
8ØCØ: D5CD 2BØØ D1FE 7FCA 7175 B7C4 Ø252 18FØ ..+.....qu...R..
8ØDØ: 21D9 6Ø35 CD85 7421 D96Ø 34C9 FD7E 34E6 !.`5..t!.`4..~4.
8ØEØ: 4ØEE 4ØC8 3EØ1 B7C9 CD86 61C1 ØAA3 C8FD @.@.>.....a.....
8ØFØ: 75ØC 7AØ7 Ø7Ø7 57ØE Ø179 A32Ø Ø514 CBØ1 u.z...W..y. ....
81ØØ: 18F7 FD71 ØD3A 8Ø38 477A Ø1ØØ FE6Ø C64Ø ...q.:.8Gz...@.@
```

CURSOR POSITIONED
OVER BYTE TO BE
CHANGED. MOVE BY
→ ← ↑ ↓ KEYS

for example, will cause a jump to RAM location **8000H** with breakpoints at locations **8010H** and **8020H**.

In TRSDOS Model III DEBUG: Type J. Answer the DEBUG response J ADDRESS?= with the same format as in the Model I case, specifying breakpoints or no breakpoint, for example:

```
J ADDRESS? = 8ØØØ,8Ø1Ø,8Ø2Ø ENTER.
```

In the TRSDOS Model III case, you may leave off the transfer address; in this case the jump will be made to the current address in the PC register. This is nice because it allows rapidly tooling through a program from one breakpointed location to another. (Sorry about the slang; it's from my Beach Boys days . . .)

8. **Tooling Through a Program (continued).** Single Stepping:

A. Another way to go through a machine language program under control of DEBUG is with the Single Step instruction.

B. Make certain the PC counter holds the proper location for the single step. If it does not, change it by altering PC in step 6 above.

C. Enter I.

D. DEBUG will execute a single instruction and effectively breakpoint. However, you cannot do this in ROM.

9. **Tooling Through a Program (even more):** Executing an entire CALL, RETURN:

A. Make certain the PC counter holds the proper location for the single step. If it does not, change it by altering PC in step 6 above.

B. Enter C. DEBUG will execute a single step if the PC points to other than a CALL or RETURN instruction. If the PC points to a CALL, the CALL will be made, the subroutine executed (whatever it is), and a breakpoint will be made on the RETURN (wherever it is). Use this when you're certain the subroutine CALLed will not bomb.

10. **Looking at execution of a foreground task.** What in Sam Hill is a foreground task? A foreground task is a time-critical function or program that interrupts a "background" or low-priority task periodically or when required. The classic foreground task in the Model I/III is the real-time clock.

A. Display the memory area associated with the foreground task, say the real-time-clock data buffer in locations **4040H** through **4046H**.

B. Press U.

C. You'll see a continuous Update of the memory area. (You'll see the RTC values incrementing up.)

11. **TRSDOS Model III only**: *To Patch a TRSDOS Disk File.* This function allows you to examine and change the contents of a disk file, to "patch" a file directly on disk. One word of warning: It is *very* easy to patch a disk file incorrectly, so proceed with caution.

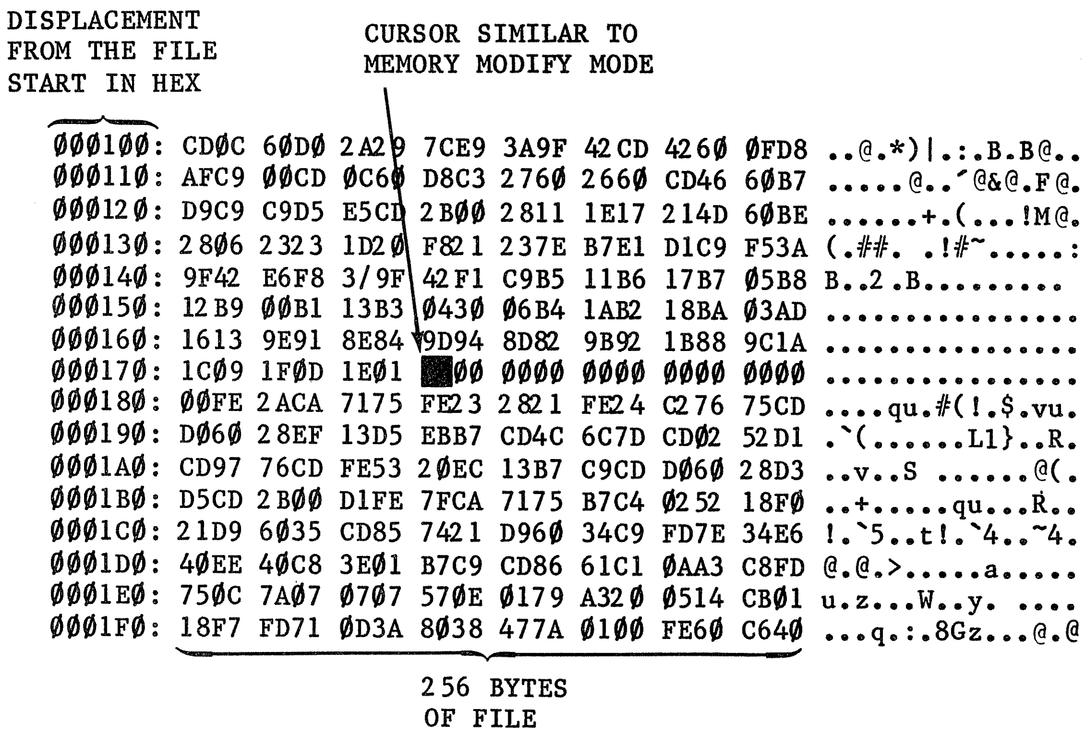
A. Load DEBUG and press F.

B. DEBUG responds with FILESPEC?

C. Enter the name of the file to be patched.

D. DEBUG will now load in the first 256 bytes of the file, as shown in Figure DT1U-5.

Figure DT1U-5 – *DEBUG Patch Action*


DISPLACEMENT FROM THE FILE START IN HEX
CURSOR SIMILAR TO MEMORY MODIFY MODE
256 BYTES OF FILE

```
ØØØ1ØØ: CDØC 6ØDØ 2A29 7CE9 3A9F 42CD 426Ø ØFD8 ..@.*)|.:.B.B@..
ØØØ11Ø: AFC9 ØØCD ØC6Ø D8C3 276Ø 266Ø CD46 6ØB7 .....@..´@&@.F@.
ØØØ12Ø: D9C9 C9D5 E5CD 2BØØ 2811 1E17 214D 6ØBE ......+.(...!M@.
ØØØ13Ø: 28Ø6 2323 1D2Ø F821 237E B7E1 D1C9 F53A (.##. .!#~.....:
ØØØ14Ø: 9F42 E6F8 3/9F 42F1 C9B5 11B6 17B7 Ø5B8 B..2.B.........
ØØØ15Ø: 12B9 ØØB1 13B3 Ø43Ø Ø6B4 1AB2 18BA Ø3AD ................
ØØØ16Ø: 1613 9E91 8E84 9D94 8D82 9B92 1B88 9C1A ................
ØØØ17Ø: 1CØ9 1FØD 1EØ1 █ØØ ØØØØ ØØØØ ØØØØ ØØØØ ................
ØØØ18Ø: ØØFE 2ACA 7175 FE23 2821 FE24 C276 75CD ....qu.#(!.$.vu.
ØØØ19Ø: DØ6Ø 28EF 13D5 EBB7 CD4C 6C7D CDØ2 52D1 .`(......L1}..R.
ØØØ1AØ: CD97 76CD FE53 2ØEC 13B7 C9CD DØ6Ø 28D3 ..v..S ......@(.
ØØØ1BØ: D5CD 2BØØ D1FE 7FCA 7175 B7C4 Ø252 18FØ ..+.....qu...R..
ØØØ1CØ: 21D9 6Ø35 CD85 7421 D96Ø 34C9 FD7E 34E6 !.`5..t!.`4..~4.
ØØØ1DØ: 4ØEE 4ØC8 3EØ1 B7C9 CD86 61C1 ØAA3 C8FD @.@.>.....a.....
ØØØ1EØ: 75ØC 7AØ7 Ø7Ø7 57ØE Ø179 A32Ø Ø514 CBØ1 u.z...W..y. ....
ØØØ1FØ: 18F7 FD71 ØD3A 8Ø38 477A Ø1ØØ FE6Ø C64Ø ...q.:.8Gz...@.@
```

E. You can now scroll back and forth through the file contents by using the ";" and "-" keys in the same way as you can scroll through memory in Step 4 above.

F. To change any location in the file:

a. Scroll through the file until you find the area to be changed.

b. Press M to enter the Modify mode.

c. Use the arrow keys to position the cursor to the hexadecimal digits to be changed.

d. With the cursor positioned over the digit to be changed, type in the new value. The digit will be changed, and the cursor will move to the next digit.

e. Repeat step d, or steps c through d to change additional data.

f. When you've changed all the data on a page, press ENTER. At this point, the data will be entered into the file. (If you've made a mistake, it is now in the disk file!).

g. To cancel changes at any point, press BREAK in lieu of ENTER for step f above. At this point the page displayed may still be changed from its original contents. Scroll back (-) and then forward (;) to restore the screen (and memory) to the original contents. (What you've done is to simply reread the data from disk.)

G. To change another file, press BREAK. This will bring you back to the F command mode, and you can load and change another file. When you're done changing files, type ENTER without a file name, and you'll be returned to TRSDOS READY.

12. **To get back to TRSDOS from DEBUG**: For TRSDOS Model I or LDOS DEBUG, enter GO ENTER. For the Model III, type Q for Quit. Once you've returned to TRSDOS on the Model I, you can turn DEBUG off to avoid inadvertent entry by

```
DEBUG (OFF)
```

13. **Other niceties**: Use H or A to set the display to H(exadecimal) or A(SCII) on TRSDOS Model I or LDOS DEBUG. TRSDOS Model III DEBUG displays in both hexadecimal and ASCII.

14. **LDOS users**: DEBUG (E) turns on the extended DEBUG with a whole host of additional commands in addition to those above. To briefly summarize:

Move Block	Baaaa,bbbb,nnnn ENTER	Moves block starting at aaaa to bbbb. Number of bytes in nnnn.
Enter Data	Eaaaa SPACE BAR	Similar to Modify. Continue by SPACE BAR, exit by ENTER.
Fill Memory	Faaaa,bbbb,cc ENTER	Fill aaaa through bbbb with cc (hex).
Jump Over	J	Jump over next byte (an I with no instruction execution).
Locate Byte	Laaaa,dd ENTER	Finds dd (hex). Searches from aaaa.
Next Load Block	Naaaa ENTER	
Ohmigosh	O ENTER	Return to LDOS.
Print Block	Paaaa,bbbb ENTER	Print a block of memory in hex and ASCII from aaaa through bbbb.
Display Port	Qii ENTER	Display byte at port ii (hex).
Write Port	Qoo,dd	Write dd (hex) to port oo
Type ASCII	Taaaa SPACE BAR	Enter ASCII characters starting at aaaa. One keyboard character at a time is converted to ASCII and entered into memory. Continue by SPACE BAR, exit by ENTER.
Compare Block	Vaaaa,bbbb,nn ENTER	Compare block at aaaa with block at bbbb for nn (hex) bytes.
Word Search	Waaaa,dddd ENTER	Search memory for dddd (hex), a 16-bit word value starting at aaaa. Memory display will show address where dddd is found.

15. **LDOS users (use with care to avoid obliterating diskettes)**: To read or write a sector *anywhere* on disk, do

```
a,b,c,d,eeee,f
```

where a is the disk drive number, b is the track number, c is the sector number to read or write, d is "R" for Read, "W" for Write, or "*" for Directory Write, eeee is the RAM memory address (destination for Read, source for Write), and f is the number of sectors to read or write.

This is a very powerful function. Use it with care to examine not only disk files, but directories, bootstrap loaders, or any disk data. Use the Write portion only if you know exactly what you are doing, as changing even 1 bit on a diskette erroneously is usually disastrous.

```
Ø,2Ø,Ø,R,8ØØØ
```

reads track 0 sector 0 from drive 0 into memory starting at **8000H.**

The N command is a special command to increment the display by load blocks.

DUFS
Diskettes, Using Flip Side

You can use the "flip side" of single-sided diskettes if you're adventurous and like to "do-it-yourself." I have used the flip side for Model I single-density type storage with few problems, but I would caution against using the flip side for Model III or other double-density applications without some testing of the diskettes, preferably with a good disk diagnostic (Stambaugh's "Floppy Doctor"). Remember, even though the flip side may look finished, it is not guaranteed.

Also, as one diskette manufacturer illustrates with a pie-charted diskette, the money spent for the "media" is a minute fraction of the total time you have invested in programs or data on the disk! Okay, so you're cheap . . .

Make a template from heavy cardboard as shown in Figure DUFS-1. Use an existing diskette and precise measurements to make the template or use the dimensions on the figure.

1. Using a soft-lead pencil, mark the sector index hole and protect notch on side A of the diskette.

2. Flip the disk over and mark the mirror image on side B.

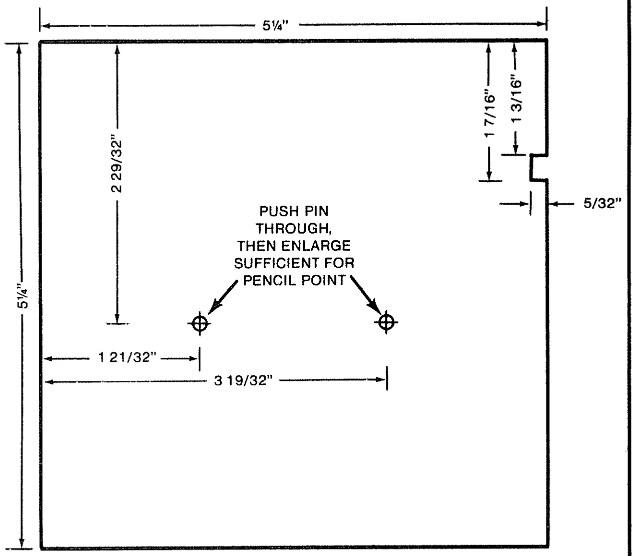
3. Take a single hole punch and clip out the write protect notch. Try to get a semi-circle.

4. Insert a small, dust-free piece of plastic with overlaying paper between the diskette jacket and diskette surface. Be careful not to scratch the surface. "Bow" the jacket to slide the two pieces in easily.

5. Using the plastic as a protective shield, slide in the hole punch so that it is positioned around the paper strip

and jacket. Punch out the sector index hole precisely centered on the pencil mark you made previously. See Figure DUFS-2.

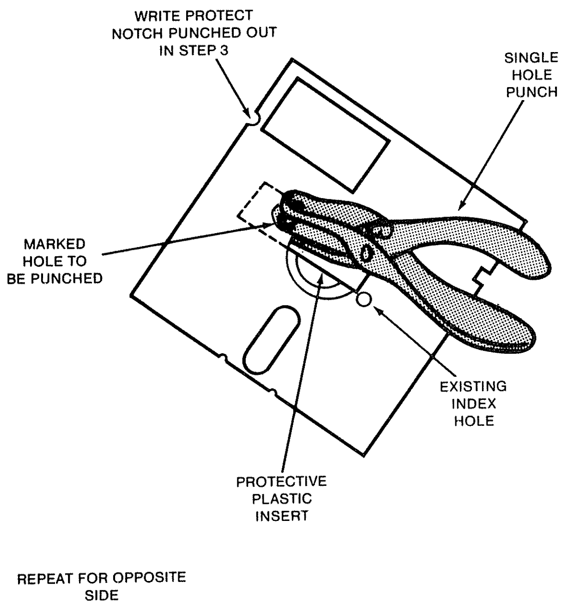
Figure DUFS-1 – *Diskette Template*



6. Turn the diskette over and perform the same operation on the other side. When you get done, you should have two clean holes that overlap each other. Try manually moving the diskette inside the surface until you can see daylight through one of the holes, through the sector index hole, and out the other hole. You should see light from one edge of the hole to the other if the two holes are positioned properly.

With a little practice, you'll be punching these out at a rate of one a minute . . .

Figure DUFS-2 – *Punching Diskette for Flip Side Use*



WRITE PROTECT NOTCH PUNCHED OUT IN STEP 3
SINGLE HOLE PUNCH
MARKED HOLE TO BE PUNCHED
EXISTING INDEX HOLE
PROTECTIVE PLASTIC INSERT
REPEAT FOR OPPOSITE SIDE

DWDK
DEBUG, Why Do I Keep Entering?, Model I/II TRSDOS, Model I/III LDOS

If you have loaded DEBUG (see DT1U) and have not entered

```
DEBUG (OFF)   (Model I)


DEBUG OFF     (Model II)
```

you will keep re-entering DEBUG every time BREAK is pressed (Model I), after a program is loaded, and upon detection of a disk-related error. Turn DEBUG OFF to avoid re-entry.

DWPR
Diskette, Write Protecting

Model I, III, Color Computer: Put a tab of opaque (solid, non-light transmitting) sticky material over the write protect hole, as shown in Figure DWPR-1.

Model III only: Also use the WP command to "software" write protect a disk drive.

```
WP (DRIVE=n)
```

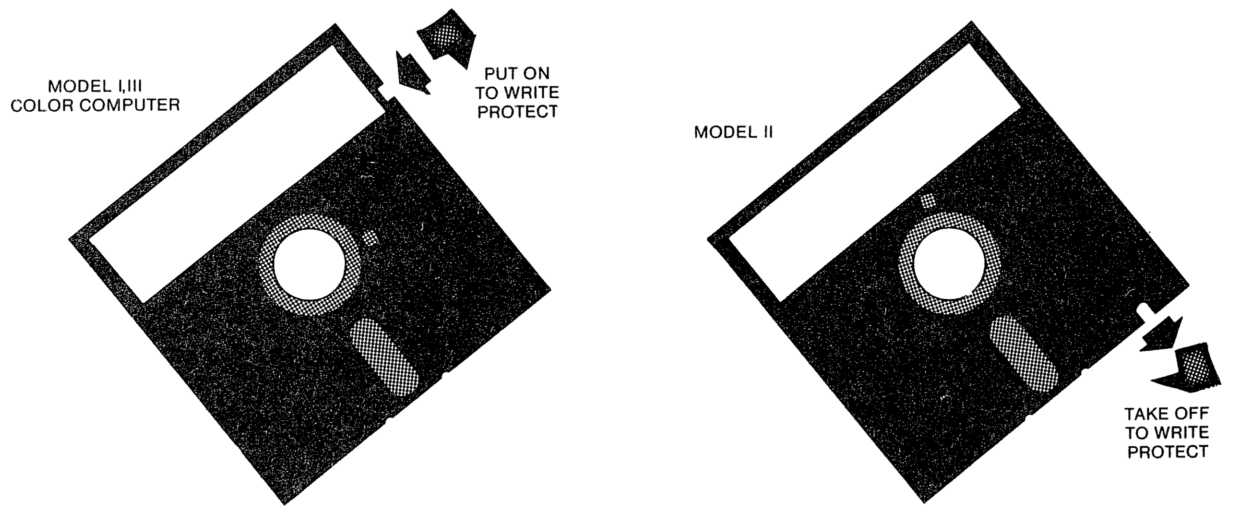
write protects drive 0, 1, 2, or 3 (one drive at a time). Entering

```
WP
```

"unprotects" the drive. The write protect tab always is operative, however, and unprotecting a drive by a WP alone will not enable writing to a tab-protected diskette.

Model II: Remove any tab from the write protect hole, as shown in Figure DWPR-1.

Figure DWPR-1 – *Write Protecting Diskette*



DZDO
Drive 0, Drive 1, Which Is It?

Model I: The drive that is closest to the computer on the cable if a standard Radio Shack cable is used.

Model II: The drive in the main unit.

Model III: The bottom drive.

Color Computer: The drive that is closest to the computer if a standard Radio Shack cable is used.

Don't feel bad, dummy, I've forgotten myself.

DZER
Divide by Zero Error

Didn't you learn anything in Algebra? If you divide a constant by successively smaller numbers, what do you get? 5/.01 = 500; 5/.00001 = 500,000; 5/.00000000001 = 500,000,000,000 ; and so forth ad infinitum. Dividing a variable by 0 is "indeterminant." (Dividing a constant by 0 produces an infinite result, math teachers please correct me if I'm wrong).

notes

EAIM

EDTASM +, Color Computer, Asembling Into Memory

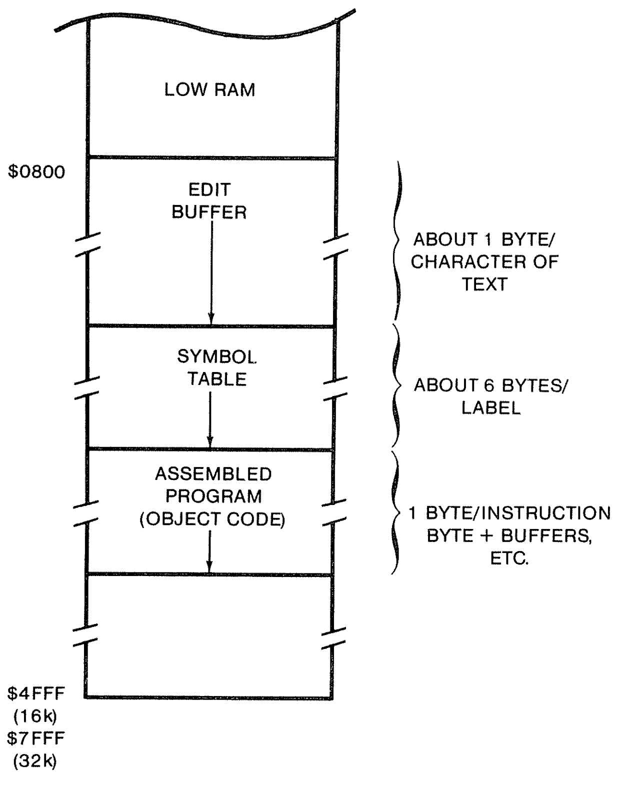
General Philosophy of in-Memory Assembly: See EDCE for general information on using EDTASM+ for editing and assembling. Here we'll discuss how to assemble directly "into memory."

In this mode the assembler translates the source code, prepared by the Edit portion of EDTASM+, into object code, but instead of saving it on cassette tape (see EDCE), it puts it directly into the Color Computer RAM. This mode is normally used as a convenience in debugging. The program is assembled directly into memory, debugged with ZBUG, and then assembled to cassette tape after a final version is hacked out. (A programmer's axiom, originated by Babbage: There are no final versions).

The in-memory assemble is done when an A/IM command is input. Associated with the /IM "switch" are two other switches, /AO, Absolute Origin, and /MO, Manual Origin. We'll get to /AO and /MO in a minute. (This whole write-up on assemblies, by the way, is kind of a sourceobject with me . . . .)

After you do an A/IM assembly, EDTASM+ will assemble the source lines and store the object after the edit buffer and "symbol table." See Figure EAIM-1. The edit buffer starts at **$0800**, and the source code is stored directly after the source lines in the edit buffer as shown in the figure. A typical starting point for a small program of 40 lines might be about **$08E0.**

Figure EAIM-1 – *EDTASM+* Edit Buffer/Symbol Table


LOW RAM
$0800
EDIT BUFFER
ABOUT 1 BYTE/ CHARACTER OF TEXT
SYMBOL TABLE
ABOUT 6 BYTES/ LABEL
ASSEMBLED PROGRAM (OBJECT CODE)
1 BYTE/INSTRUCTION BYTE + BUFFERS, ETC.
$4FFF (16k)
$7FFF (32k)

Assembling in Memory With Floating Origin: In many cases your source program won't have any Origin statements, and you'll want to let EDTASM+ store the object code right after the Edit Buffer. Why? Maximum use of memory, and you won't have to worry about the origin. But how will you know where the program is? Easy. Use a symbol for the first statement of your program. Suppose that the first line of your program is

```
START   LDX     #$123E          load buffer address
```

After you do the A/IM, you'll see a symbol table printout with an entry for START, such as

```
START   Ø8EØ
```

You'll know that the program starts at **$08E0.** You can then use the "symbolic addressing" mode of ZBUG to access the program by symbols such as START, any other program symbols, or their absolute location. ZBUG refers to an assembler "symbol table" to find labels in the program and match them to their absolute locations. More on this in the ZBUG procedure ZUEC.

Assembling in Memory with Absolute Origin: After you've done a few assemblies, you'll know what areas of RAM are available. In fact, all of RAM from the end of the Edit buffer to top of memory (16K, 32K, or more, if you have one of those 64K systems) is available for your program. You can assemble at any available area by using the /AO, Absolute Origin switch. You'd have something like:

```
*A/IM/AO
```

In addition to the /AO switch, you'd need an ORG pseudo-op (see POCE) in your source code, usually at the very beginning. You might have something like:

```
        ORG     $ØAØØ       set ORG
START   LDX     #$123A      start of buffer
```

After the assembly, you'd see that EDTASM+ loaded the object starting at location **$0A00**, the location defined by the ORG. Why use /AO and ORG? I like to start programs or sections of programs at "even boundaries" like **$1000.** It makes finding the locations of things so much easier Even though you can still use ZBUG to find the location of a buffer by referencing "BUFFER," it's reassuring for a paranoid like myself to be able to find it in the same location as is on the assembly listing. If you ORG a program at **$1000** and then use the /AO switch, the program listing will start at **$1000**, and you can find program locations directly from the listing without using symbolic references.

What happens If You Have an ORG without /AO?: If you have an ORG in your program and assemble without /AO, you'll see a strange result. The location of the program will become the ORG value + the EDTASM+ object code location. As an example, if you have ORGed

```
       ORG    $1ØØØ        set ORG
START  LDX    #$123A       set buffer loc
```

and you do not set /AO, EDTASM+ would add **$1000** and the normal program location of, say **$0800**, to give a location of **$1800** for START. Why? Basically this is because EDTASM+ thinks that you don't know what you're doing. If EDTASM+ did not do this, then ORGs that allocate buffers and space (in lieu of RMB, see POCE) such as

```
BUFFER   EQU    *        buffer
         ORG    *+2ØØ    2ØØ bytes
```

would not assemble correctly. Therefore, use /AO if you have an initial ORG for program location, but not for "space allocation" ORGs.

What the Heck is /MO?: The /MO switch is used to give you even more control over in-memory assemblies. /AO lets you assemble anywhere in "user RAM" between the Edit Buffer/Symbol Table and top of memory. (Remember that EDTASM+ itself is in the **$C000** area — see MMCC).

E

/AO does not let you move the Edit Buffer/Symbol Table around, however. /MO lets you set the address of the Edit Buffer/Symbol Table as well as letting you set the location of the Origin of the assembled program. Why? Why not? EDTASM+ is an offshoot of EDTASM+ for the Z-80 (Model I), and it gave you these controls. Although normally you would not want to do this, the capability is there. (As soon as I say that this is not too useful, someone will write in telling me what an idiot I am; therefore, I will be as vague about /MO use as the EDTASM+ manual is. But seriously, you may have variable storage in the memory area occupied by the EB/ST — no, that doesn't sound realistic — you may have graphics page data in that area — no, you can reset the graphics page pointer — well, anyway, it's a damn good idea . . .)

In any event, to use /MO:

A. Set the location of Edit Buffer/Symbol Table by changing an EDTASM+ location called BEGTMP at location **$00FF**. This location contains the *most significant byte* of the Edit Buffer/Symbol Table address minus about **$0200**. Initially it is a **$06**, which puts the EB/ST at about location **$0800**. Put in any value from **$06** up, and you will define the EB/ST starting location minus about **$0200**. Note that changing the ms byte changes USRORG by "page boundaries" or multiples of 256 — **$0600, $0700, $0800**, etc).

B. Set the location of the assembled object code by changing an EDTASM+ location called USRORG in locations **$00FD, E**. Changing this variable will set the upper limit that EDTASM+ can use for the EB/ST and will force EDTASM+ to start assembling object code at that location.

C. Delete any ORG commands that define program locations at absolute locations, otherwise you'll have the same difficulties as in /AO above. You can leave ORGs that "reserve space" as in

```
BUFFER   EQU    *
         ORG    *+2ØØ
```

D. Assemble using A/MO. Voila, Monsieur! Zee manuelle oreegan . . .

ECCE
Expressions, Color Computer EDTASM+

There are many arithmetic operators that can be used in source code or in ZBUG; both sections of EDTASM+ use the same format and expression evaluation. ZBUG also has a calculator mode that lets you use it to display the results of expressions like

```
1ØØØ+SIZE+1ØT=
```

Typing in the above example would cause ZBUG to print out the result; it would evaluate any symbol from the symbol table as it did so.

How to find the value of a symbol: If you've entered ZBUG and don't know where your program object is, enter a known symbol from the program listing; ZBUG will respond with the value, which is the current location:

```
#START=88A
```

Addition and subtraction: Use plus and minus in any combination

```
START+2ØT+1ØØØ=1A4D              (ZBUG)
TABLE     FDB    DATA+23Ø-SIZE   (Assembler)
```

Multiplication and division: Use asterisk and .DIV. (.DIV. used as slash means "open a location" in ZBUG).

```
#255T*255T=ØFEØ1              (ZBUG)
#1ØØØT*1ØØØT=424Ø             (ZBUG 1696Ø decimal)
#1ØØØT.DIV.1Ø=3E              (ZBUG 1ØØØ/16=62)
TABLE     FDB    DATA*4Ø      (Assembler)
TABLE     FDB    1ØØØ.DIV.1Ø  (Assembler)
```

Note that multiplication and division are unsigned integer operations. Multiplying 1000 by 1000, for example, yields the portion of the product which can be held in 16 bits, 16960. Dividing 1000/16 yields the integer portion of 62.5.

Modulus operations: Use the .MOD. operator. A modulus operation finds the "remainder" of a divide.

```
#1ØØØT.MOD.16T=8           (ZBUG 1ØØØ/16=62Q, R8)
TABLE    FDB   1ØØØ.MOD.16   (Assembler)
```

Positive and negative numbers: Use the plus and minus sign as you might expect. These are "unary" operators that require one operand, as opposed to "binary" operators that require two operands. Just thought you might like to know — I'm not trying to sound impressive . . .

The assembler assembles negative data as two's complement numbers (see TCNU):

```
TABLE    FCB   -67
```

for example, assembles as **$BD**, the two's complement form of -67.

Relational operators: EDTASM+ has two relational operators, equals (.EQU.) and not equal (.NEQ.). The resuit is either **$FFFF** (true) or **0** (false). Not too many applications here, although the operators could be used to set flags:

```
TABLE   FCB    PRNTR.EQU.YES    -1 if prntr, Ø if not
```

This is one of those gray areas where a writer says, "Shall I be vague about the description and not let on that I'm not sure why they included this operator?" Okay, I'll take a stand. Cheap to throw in, but not very useful . . .

Shift operators: The less than sign (<) is used to represent a shift. If the less than sign is followed by a positive value, the shift is to the left; if by a negative value, the shift is to the right:

```
TABLE   FDB    1ØØØ<-8     get ls byte
        FDB    1ØØØ<8      get ms byte
```

The two values produced above would be **$0003** and **$E800**. These are handy operators for finding address bytes or aligning data. The shifts are "logical" shifts which shift in zeros rather than recirculating data.

Logical operators: Logical AND (.AND.), logical OR (.OR.), exclusive OR (.XOR.), and complement (.NOT.) work the way they do in BASIC:

```
#5.AND.1=1
#5.OR.2=7
#5.XOR.4=1
#.NOT.5=ØFFFA
```

Notice that .NOT. is a unary operator that requires one operand.

All in all, an excellent assembler/ZBUG package that allows a great deal of flexibility in examining data and constructing assembly-language data structures and addresses.

ECIS
Embedded Codes in SCRIPSIT

If you're an LDOS user, you can also use the KSM function and BUILD to build a file of HEX data (use the BUILD HEX option) to insert non-printable characters at a single keystroke.

EDAN
EDAS, Notes

EDAS is an Editor/Assembler package for the Model I/III by MISOSYS. It contains similar commands to the Series I Editor/Assembler by Radio Shack. Read S1EA as an introduction, and I'll note the differences. In general, EDAS offers more expression operators (see AEU1), move block and more editing options, in-memory assembly, and macro-like *GET files. The latest EDAS also provides true "Macro" capability, a way of generating in-line source code.

Loading EDAS: Use the DOS command

```
EDAS (MEM=nnnn,PROMPT,JCL)
```

where MEM=nnnn protects high memory; nnnn is a decimal value, while X'nnnn' is hexadecimal. Use PROMPT for an EDAS prompt before printer page ejection. Use JCL to enable input of EDAS commands for JCL on LDOS and other systems (no source text can be entered from JCL).

Command Syntax: Use dashes for the switches instead of commas. For example, assemble by A-NO-LP. Use a comma to specify a range of lines (P300,500).

Additional Operators: In addition to addition (+), subtraction and negation (-), logical AND (&), and shift left or write (<), you can multiply (*), divide (/), find the modulus (%), logical OR (!), and logical XOR (#). Use of these is shown in Figure EDAN-1.

Additional Pseudo-Ops: In addition to ORG, EQU, DEFL, END, DEFB, DEFW, DEFS, DEFM, *LIST ON, and *LIST OFF, EDAS adds these pseudo-ops:

TITLE: titles the listing with up to 28 characters. One TITLE is accepted.

SUBTTL: subtitles the page with up to 80 characters. Use any time. Subtitle appears on next page after SUBTTL.

PAGE: forces new page.

COM: generates an ASCII comment string of up to 128 characters within the object code. Format is COM <comment string>.

SPACE: spaces n lines. SPACE 5 inserts 5 blank lines in listing.

ERR: forces error message. Format is ERR message. Use in a conditional assembly block (if conditional block is assembled when it shouldn't be, an error message is listed).

DEFB or DB: same use, but you can now have a string of items separated by commas.

DEFW or DW: same use, but you can now have a string of items separated by commas.

DEFS or DS: DS permitted.

DEFM or DM: same use, but you can now have a string of items separated by commas.

IF, ENDIF: you can conditionally assemble blocks of code by surrounding the block of source code with an IF and ENDIF:

```
MODI    EQU     Ø          ;1 if Mod I, Ø if Mod III
        IF      MODI       ;start of Mod I IF
        LD      A,5        ;body of code
        ENDIF              ;end of Mod I IF
        IF      1-MODI     ;Ø if Mod I, 1 if Mod III
        LD      A,6        ;body of code
        ENDIF              ;end of Mod III IF
```

In the above example, Mod I code assembles if MODI is non-zero while Mod III code assembles if MODI is zero value. You can also have IFEQ, IFGT, IFLT where you can compare two parameters, as in

```
IFLT   $,8ØØØH   'assemble if $ < 8ØØØH
```

***GET**: The *GET file command gets a source code file from disk and inserts it into the *GET point in the source code file. Use *GET to bring in predefined "code segments" (similar to macros) that will be inserted into source code.

Editor Commands: D(elete), E(dit), F(ind), H(ardcopy), I(nsert), L(oad), (re)N(umber), P(rint), R(eplace), W(rite) are similar, except for line ranges using commas in place of colons. Up arrow and down arrow scroll the source code display.

B now branches to DOS or to address in B nnnn.

C(hange) does a global replace of string 1 with string 2 (very handy!). Format is

```
C/string1/string2/line#1,line#2
```

where line #1 is the first line to change and line #2 is the last line to change.

K(ill) does a file KILL from within EDAS.

```
KILL KLUDGE
```

kills file KLUDGE/ASM.

M(ove) moves a block of source lines from one area of the source buffer to another. The format is

```
M line#1,line#2,line#3
```

Figure EDAN-1 – *EDAS Operators*

```
                 ØØ1ØØ ; EDAS OPERATORS
8123             ØØ11Ø          ORG     8123H
ØØØ3             ØØ12Ø SWITCH   EQU     3
ØØØ3             ØØ13Ø OP1      EQU     +3
FFFC             ØØ14Ø OP2      EQU     -4
8123 7E          ØØ15Ø TABLE    DEFB    1ØØ+23+SWITCH  ;ADD/SUB
8124 E8Ø3        ØØ16Ø          DEFW    +1ØØØ          ;SIGN
8126 18FC        ØØ17Ø          DEFW    -1ØØØ          ;SIGN
8128 2Ø81        ØØ18Ø          DEFW    TABLE&ØFFFØH   ;LOGICAL AND
812A ØØ23        ØØ19Ø          DEFW    TABLE<8        ;LEFT SHIFT
812C 81ØØ        ØØ2ØØ          DEFW    TABLE<-8       ;RIGHT SHIFT
812E 84Ø3        ØØ21Ø          DEFW    1ØØ*SWITCH*3   ;MULTIPLICATION
813Ø ØB2B        ØØ22Ø          DEFW    TABLE/3        ;DIVISION
8132 23ØØ        ØØ23Ø          DEFW    TABLE.MOD.64   ;MODULO
8134 2381        ØØ24Ø          DEFW    TABLE!32768    ;OR
8136 FFFF        ØØ25Ø          DEFW    OP1.XOR.OP2    ;EXCLUSIVE OR
ØØØØ             ØØ26Ø          END
ØØØØØ Total errors
```

where line #1 is the first line in the block to be moved, line #2 is the last line in the block to be moved, and line #3 is the line that the block should follow.

```
M 35Ø,37Ø,99Ø
```

moves lines 350 through 370 after line 990.

Q(uery) is a Disk DIR from within EDAS. The format is either Q or Qn, where n is the optional drive number.

S(witch) switches from upper case to upper/lower case.

T(ype) is identical to H except that line numbers are not printed.

U(sage) displays the number of bytes of text buffer in use, the number available and the first address available for in-memory assembly.

V(iew) is a LIST file command from within EDAS. The file will be listed but not loaded into the buffer area. Use to view a file before loading.

(e)X(tend) extends the text buffer area by destroying the assembler in memory. Use with large source files.

1: Entering n1 n2 sets the number of lines to print per page to n1 and the page length in lines to n2.

Assembler Commands: -NL, -LP, -WE work as in the Series 1 EDTASM.

-WS (with symbol) enables symbol table display and printing. Default is no symbol table.

-NE surpresses DEFM/DM, DEFB/DB, and DEFW/DW expansions on listings (prints only one line).

-XR generates a cross reference listing file on disk.

-IM assembles code directly to memory on the basis of your ORG pseudo-ops.

-WO assembles object code to a disk file.

The format of A is

```
A filespec1,filespec2-XX-XX...
```

where filespec1 is the object file (if necessary) and filespec2 is the optional cross reference file.

Special format files: W-writes the source file without a 6-character file name header. W # writes the source file without line numbers. W-# deletes both. L-, L#, and L-# operate in the same manner on loading the source file.

Macro capability: Macros are a way of generating from one to dozens of lines (or more) of source code with a single macro "call." The newest EDAS, EDAS IV, provides macro capability, and even nested macros. If you are or want to be a serious assembly language programmer, I would heartily recommend EDAS IV. Even without macro capability it is an order of magnitude better than the RS Series I editor/assembler, and with macro capability it is terrific. (You should hear me when I'm *really* enthusiastic . . . .)

EDCE

EDTASM+, Color Computer, Using Editor and Assembler

EDTASM+ is a 6809E Editor/Assembler/Debug package available as a ROM program for 16K machines with cassette. You can edit and assemble 6809E assembly language programs with EDTASM+ (see ALWI for a description of assembly language) and it also has a comprehensive debug capability.

To Load EDTASM+:

1. Turn off the Color Computer.

2. Load the ROM pack by inserting it right side up into the right-hand side of the Color Computer. You should hear a definite "tonk" as the ROM pack snaps into place.

3. You should see the title message and prompt:

```
EDTASM+ 1.X
COPYRIGHT c 1981 BY MICROSOFT
```

General Description:

The EDTASM+ package consists of three parts, the Editor, the Assembler and a debug package called ZBUG.

The Editor is used to construct or modify "source" files that are largely ASCII files (see AFWA) with the exception of some non-standard characters for line numbers. The source files are resident in RAM while they're being edited but can be stored on cassette for later use.

The Assembler is used to "assemble" (see ALWI) the source files in RAM. This consists of translating the symbolic source code representing 6809E machine language instructions into an "object file". The object file is largely machine language codes (see MLWI) with minor "header information" that indicates the area of RAM to be loaded and other data.

The object file, when properly assembled, represents a machine language program that can be loaded from cassette (see LEMC) and executed. The object "code" may be anywhere from several instructions to thousands of instructions long and is dependent upon the application. The object code can also be assembled in

RAM for immediate execution. Normally this is part of the debug process before writing out a final version as a cassette file.

Before proceding, you must know something about 6809 assembly language. See ALWI.

To Create a New Source Program:

1. While in the Command Mode, as indicated by the "*", type I. This puts you into the Line Insert Mode, as shown in Figure EDCE-1, starting with line number 00100. 00100.

Figure EDCE-1 — *Editor/Assembler Display*

```
EDTASM+ 1.Ø
COPYRIGHT (C) 1981 BY MICROSOFT

*I
ØØ1ØØ█
                    CURSOR AT FIRST CHARACTER
                    POSITION FOR LINE ØØ1ØØ
```

2. Enter your source lines one at a time. While entering your lines, you can use the Edit Mode subcommands. These are virtually identical to the BASIC Edit Mode subcommands described in EMBH and allow you to edit characters within a line. Each line is terminated with an ENTER.

Note: A *range* of lines can be specified in the Editing commands by using an exclamation mark. 100! 5 means the same as the 5 lines starting at line 100.

3. When you have entered the last line, type BREAK to get back to the Command Mode. You can now list and correct the source lines, save the lines as a source file, or perform other Edit actions. We'll describe each in turn.

To List and Correct Lines and Perform Other Line Edit Actions:

Besides the Edit Mode subcommands that operate on characters within lines, you can also delete, modify, or add lines to the source file. All of these commands work while in the Command Mode, as indicated by the asterisk (*) prompt.

1. **To Edit any line** on a character basis, enter En ENTER, where n is the line number. To Edit line 1100, for example, you'd enter

```
E11ØØ
```

to get back into the Edit Mode described above.

2. **To Print (display)** the source lines, enter Pn:m, where n is the starting line number and m is the ending line number. To display lines 300 through 1060, for example, you'd enter

```
P3ØØ:1Ø6Ø
```

Display one line by a Pn. Display the first line by P#. Display the last line by P*. Display the current line by P(period).

3. **To get a hardcopy listing** use the H(ardcopy) command for your system line printer in the same formats as 2. To print lines 400 through 450, for example, you'd enter

```
H4ØØ:45Ø
```

To get a hardcopy without line numbers (text only) use a T in place of the H.

4. **To delete a line** or range of lines, enter Dn or Dn:m, where n is the starting line to be deleted and m is the ending line. The starting line must exist, otherwise you'll get an error message.

5. **To insert a line**, use the In command, where n is the line number of the insert line. The line number generally follows the line number immediately before the insert point. To insert a source line directly after line 1010 and before the following line 1020, you'd enter

```
I1Ø11
```

To insert a series of lines, use the I command. EDTASM will automatically renumber the lines as new lines are added. All lines will be inserted at the same point. If you'd like to avoid the renumbering, use the In,m form of the insert, where m is a line increment. If the existing lines increment by 10, for example, you'd be able to insert 9 lines between lines 1010 and 1020 without renumbering by

```
I1Ø11,1
```

6. **To renumber the lines**, use the Nn,m command, where n is the starting line number (often 100) for renumbering, and m is the increment (often 10).

7. **To replace a line** and continue in the line insert mode, use the Rn command, where n is the line number of the line to be replaced. The line will be replaced with the next entered line, and from that point the operation will be identical to the line insert mode described in 5 above.

8. **To find a text string.** Use the Fxxxx command, where xxxx is a text string to be found. Handy for locating lines by looking for "labels." If you want to look for a series of labels, enter Fxxxx and follow with F alone. The command will use the previous search string.

9. **To copy a block of lines,** use the C command. The format is C,newline#, startline#, endline#, increment. To copy lines 300 through 400 to a new area starting with line number 1000 and with increment of 10, for example, you'd have:

```
C1ØØØ,3ØØ:4ØØ,1Ø
```

At the end of the Copy, lines 300 through 400 would be reproduced in totality at the 1000 area. Lines 300 through 400 would remain unchanged.

10. **To Move a block of lines**, use the M command. Move operates similarly to Copy, except that the original block of lines is deleted. To move lines 300 through 400 to a new area at 1000 with increment of 10, for example, you'd have:

```
M1000,300:400,10
```

11. **To go to ZBUG, type Z.** For ZBUG commands and operation, see ZUEC.

12. **To return to BASIC**, enter Q. Type EXEC 49152 to return to the Editor from BASIC.

13. **To scroll up or down** one edit line, press up or down arrow.

To Write a Source File:

After you have edited your source lines by the Line Edit and Character Edit commands above, you can either assemble the file (see below) or write out the source file to cassette. To write to cassette:

1. Enter W name, where name is the name of the file. If no name is specified, the name NONAME will be used.

2. You'll now see a READY CASSETTE message. Prepare the cassette and press any key. A cassette write operation will proceed until the entire source is written.

3. After the write, the asterisk (*) prompt will be displayed.

4. Verify the source file on cassette by using the V command. V operates identically to the BASIC SKIPF command, simply comparing the data on the file with the data in memory to make certain it is valid. Enter V name to verify the cassette file just written.

To Load an Old Source File:

An old source file can be loaded for assembly or modification by this procedure:

1. Delete any source file in memory by D #:*. If you do not do this, the load will append the new file from cassette onto the old source lines.

2. While in the command mode (asterisk prompt), enter Lname, or simply L if the next file is the one desired.

3. If you are loading from cassette, EDTASM+ will prompt you to prepare the cassette. If you do not enter a name, NONAME will be used.

To Assemble:

Once you have a good source file in proper format, you can assemble. Assembling translates the source code lines into equivalent machine language code (see ALWI). You can assemble with the following commands and options:

1. **To assemble without an object file to cassette:**

A. While in Command Mode (*), enter A/NO.

B. You'll see a listing rapidly displayed on the screen.

C. At the end of the listing you'll see XX TOTAL ERRORS, but it may go by too fast too observe.

D. You'll then see a "Symbol Table" display.

2. **To assemble without an object file and with no symbol table listing:**

A. Enter A/NO/NS.

B. You'll see a listing and error indication with no symbol table. Any errors will have been displayed too fast to catch.

3. **To assemble without an object file and to "wait on errors":**

A. Enter A/NO/WE or A/NO/NS/WE, depending upon whether you want a symbol table listing (NS is No Symbol Table).

B. The display will stop as each assembly error comes up. Press any key to continue.

C. You'll get the error message at the end.

4. **To assemble to line printer without object:**

A. Use any of the commands above, but add /LP to the end of the command line.

B. You'll get a simultaneous display and line printer listing.

5. **To get object output on cassette:**

A. If you use any of the above command sequences without the /NO option, you'll get "object" output to cassette, with a NONAME name. If you use the format A name/XX/XX/XX, where XX are options (switches), you'll get an object output to cassette.

6. **To load the object file and execute,** see LEMC.

7. **To assemble directly into memory,** use any of the above options, do not use a file name, and specify the option /IM, for in-memory assembly:

A. Use either /AO or /MO, absolute or manual origin options for in-memory assembly. Since in-memory assembly is normally associated with ZBUG and debugging, we'll explain the ORiGin problem in EAIM.

For more information on assembler operations, see:

ALWI for general information on assembly language

POCE for information on assembler pseudo-ops

AECE for assembler errors

ECCE for assembler expressions

ZUEC for ZBUG operations and in-memory assembly

EMBH
Edit Mode, BASIC, Most Systems

(Does not apply to Model I/III, Level I or Color Computer Color BASIC. Edit by replacing lines on these machines. See RLIB).

The Edit mode in BASIC allows you to change, insert, or delete characters from a BASIC line. It is not too handy in BASIC statements such as "100 A=0", but is terrific for BASIC statements such as

```
"1ØØØ A=(I-3)*(A(I)-B(I))/256.7: IF (A<7 OR A>19)
THEN GOTO 2ØØØ ELSE IF A=1Ø THEN PRINT "SELL TANDY
STOCK"."
```

E

To enter the Edit mode, type

```
EDIT line#
```

while in the command mode. You may also type "EDIT." to edit the "current line" (except for Color Computer).

Once you're in the Edit mode, you can *move the cursor back and forth* by the space bar and left arrow (or backspace) keys. The cursor is always over the character on the line to be edited.

To move multiple spaces right, enter one or two digits followed by space bar. The cursor will move right the number entered. Entering 5 space bar, for example, moves 5 character positions right.

To move multiple spaces left do the same with the left arrow or backspace. Entering 5 left arrow moves 5 character positions left.

To delete the current character (at the cursor position), type "D". To delete multiple characters, enter one or two digits followed by "D".

To change a single character, type "C" followed by the character to replace the one at the cursor position. *To change multiple characters*, type one or two digits, followed by "C", followed by the new text.

To insert characters, type "I". You're now in the Insert mode. To get out at any time, type SHIFT, up arrow (ESC on Model II). All characters typed before a SHIFT, up arrow will be inserted before the cursor position. There are *certain other commands* that also place you in the Insert mode after performing other actions:

A. *Entering "H" "hacks off"* the remainder of the line from the cursor position and puts you into Insert at the hack point. (Here's your chance to be a hack writer).

B. *Entering "X" displays* the remainder of the line and puts you into Insert at the end of the line.

To search for a given character, type "S", followed by the character. Edit mode will find the character and position the cursor at the character position. Typing one or two digits followed by "S" will find the "nth" occurrence of the following character. Entering 5SX, for example, finds the 5th X from the current cursor position. **"K"** is like S, except that it *"kills,"* or deletes, the entire line up to the point at which the character is found. ("Hack," "Kill," — sounds like a Jamie Lee Curtis movie).

To redisplay the entire line: Type "L".

To ENTER the line after all changes have been made and end the Edit, type "E" or ENTER.

To cancel all changes that have been made and restart the Edit, type 'A'.

To cancel changes and quit the Edit, type "Q".

EPHT
Ending a BASIC Program, How to, All Systems

Your last BASIC statement should be an END, as in

```
1ØØØ END
```

However, you really don't need an END, provided that the program ends with the last statement. For an example of trouble:

```
1ØØØ PRINT "TH..TH..TH...THAT'S ALL FOLKS!"   'end
1ØØØØ PRINT I,J
1Ø1ØØ RETURN
```

should end at line 1000, which is the last program statement you want executed. However, when the BASIC interpreter gets to 1000, it continues on to the subroutine at line 10000 which will give a "Return Without Gosub" error. In this case you need an END at line 1010.

EPWT
Electronic Parts, Where to Get

Let's face it, I want this book to be sold to Radio Shack. In spite of that, however, Radio Shack remains one of the best places to get common electronic parts of all types. Try Radio Shack first for some integrated circuits (not a great selection), all electronic tools, multimeters, continuity testers, connectors (including RS-232-C), wire, cable, resistors, capacitors and the like.

For less common electronic parts, you'll have to leave the resonant echo of "How May I Help You?" and go to a store where they don't demand a mailing list address at the top of the sales slip. If you know a radio amateur or electronics experimenter, ask him where a good electronics parts store is. Almost every town over 30,000 or so has a good electronics store where they carry every conceivable type of component; although, again, they'll be light on integrated circuits. Expect to pay slightly more than at Radio Shack.

For integrated circuits, find one of the places that specializes in ICs, and not the other components. This should not be a problem if you live in a major metropolitan area, but will definitely be a problem in smaller towns. If you can't find such a place, look in the back pages of *Computers and Electronics, Radio Electronics,* or *BYTE* magazines. You'll find plenty of IC parts houses that, in some cases, will deliver in a few days. Prices will be as good or better than Radio Shack's.

Another alternative: For hard-to-get electronic parts, go directly to a "distributor." Distributors are representatives of the integrated circuit manufacturers (such as Motorola or Zilog) — again, in major metropolitan areas. If you know the name of the actual parts manufacturer, call a local sales office and ask for the name of a distributor for the parts. Call them up and ask for the order desk. Pretend you're an electronics consultant, and put the parts you need on "will call," a buzz word that means that you will walk in and pick them up.

They'll ask for your purchase order for the parts. Give them a dummy number, such as "V030233," or simply tell them that it is "verbal," a buzz word for "no purchase order and I'm trying to pretend I'm an electronics consultant, but I know you don't believe me." Unless you have a state tax resale number, tell them the parts are "not for resale" when they ask.

You'll pay more for the parts through a distributor, but at least you'll be able to get them without delay.

ETIB
Error Trapping in BASIC, Some Systems

Model I Level II and Disk, Model II, and Model III Level III and Disk BASIC have error-trapping capability, represented by the "ON ERROR GOTO," "RESUME," "ERROR," "ERR," and "ERL" BASIC commands. The Model II also has a sixth command, ERR$. What is error trapping? How do you use it? Why am I asking all of these rhetorical questions?

Error trapping gives your program complete control over error conditions. Suppose you have invested a good deal of time in generating an "idiot-proof" pork bellies commodity program to be run by your niece. Now, your niece knows nothing about pork bellies and even less about your TRS-80. Without error trapping, she'll get a "Write Protected Disk" error message, and the program will stop. With error trapping, you can build in a "Take the tab off the diskette, dummy!" message, she can take the corrective action, and the program can proceed.

The basic steps in incorporating "error processing" in your program are these:

1. Code up an error-processing subroutine. This error-processing subroutine can be anywhere in your BASIC program. If you have enabled error trapping (by the "ON ERROR GOTO" command we'll discuss later), this subroutine will be automatically entered.

2. In the subroutine, you can find the type of error by using the ERL and ERR functions. ERL returns the line number in which the error occurred, and ERR returns the error code. Use ERR/2+1 to find the true error code for the Models I and III. The error codes can be found in the back of your BASIC or Disk BASIC manual.

3. You'll want to use ERR (or ERR/2+1) to find the type of error code. Some of these error codes will be conditions about which you can do nothing in your program. For example, if you get an "Undefined Error," throw up your hands and give up to BASIC (which we'll show you how to do in RESUME). Other error codes can be handled, such as "Write Protected Disk," or "Disk Full." In the latter cases, print a prompt message or take a corrective action within your program. ERL will return the line in which the error occurred, so you can make certain it is an expected condition. Use ERL and ERR just as you would other functions, such as 10000 CD=ERR/2+1: LN=ERL.

4. When you want the program to enable error trapping, execute an "ON ERROR GOTO XXXXX" command, where XXXXX is the line number of the error processing routine. Any error that occurs after error trapping has been enabled will cause your error processing routine at XXXXX to be entered.

5. During certain parts of your program, you may not want error trapping. If your error processing subroutine handles only disk errors, for example, there's no need for error trapping during heavy string manipulations. In this case, incorporate an "ON ERROR GOTO 0" command to disable error trapping. In this case any error will cause a return to the BASIC interpreter and a stop.

6. The last command in your error trapping subroutine should be a "RESUME" command. There are three types of RESUME. RESUME NEXT resumes execution of the program at the line following the error line; presumably

your error processing subroutine has handled the error condition. RESUME with a line number, such as "RESUME 13000," resumes execution at the specified line number. In this case you may have branched to another action because of the error and/or corrective action. RESUME without a line number or with a line number of 0 causes re-execution of the line in which the error occurred. In this last case, unless you've disabled the error trapping by an "ON ERROR GOTO 0", you'll be right back at the error processing again; presumably, in this case you're throwing up your hands.

7. Model II users only: ERR$ returns a descriptive message about the error type. This function is a way of further defining a TRSDOS error occurring in BASIC.

Ok, got it? Code up an error processing subroutine terminated by a RESTORE. Enable and disable this error trapping by ON ERROR GOTO XXXXX or ON ERROR GOTO 0. Check for recoverable conditions within the error processing by ERL and ERR.

One final word, use the ERROR function to "simulate" the error. This function allows you to test the error trapping code without having to wait for an actual program or operator error. A

```
1ØØ ERROR 1Ø
```

for example, simulates a redimensioned array error and causes a transfer to your error-processing logic.

Good luck with the pork bellies.

EYBP
Erasing Your BASIC Program, All Systems

Enter NEW. Voila! Your program, she is fini! NEW "reinitializes" the BASIC interpreter "pointers," in effect erasing your program.

FBDF
Functions, BASIC, Defining, Some Systems

Model I/III Disk BASIC, Model II, Color Computer Extended and Disk BASIC only: The DEF FN command is used to define functions. What is a function? A function is a commonly used operation in BASIC that is not part of the BASIC set of commands. BASIC has many built-in functions (generally a command that requires parentheses around its argument), such as SQR(), POKE(), and INP(). You can also define your own functions with DEF FN.

Suppose you found that in your BASIC program you used the formula for finding the future amount of an investment P left in an interest account at interest I per annum for N years:

```
S=P*(1+(I/12)) ↑ (N*12)
```

Suppose also that you used this formula 103 times throughout your program. You could define a function FNS that would perform that calculation by

```
100 DEF FNS(P,I,N)=P*(1+(I/12)) ↑ (N*12)
```

You could now replace every occurrence of the formula with

```
1000 S=FNS(P,I,N)
```

where P, I, and N would change according to the current values. For example, you might have

```
1000 S=FNS(1000,.18,12)    '$1000 at 18% for 12 mos
      •
      •
2000 S=FNS(5000,.12,13)    '$5000 at 12% for 13 mos
      •
      •
3000 S=FNS(WZ,C,14)        '$WZ at C for 14 mos
```

A function helps you define commonly used operations as another command in BASIC; it saves entering the formula each time.

Up to 26 functions can be defined by using DEF FNA through DEF FNZ. The "arguments" for the function are defined in the DEF FN statement as in the example above. You can use a number of arguments in the Model I,II or III, but only 1 in the Color Computer. The arguments are defined in the DEF FN statement as dummies; the variables used in the DEF FN are only "place markers" for the subsequent definition, and are not related to any actual variable names. The arguments used in the function "call" (the FNX( )) can be constants, variables, or expressions.

FBWT
500-Baud and 1500-Baud, for Model III Cassette, BASIC, When to Use

Use 1500-baud for all operations except reading in Model I tapes. The 1500-baud rate is inherently more reliable than the 500-baud rate. The 1500-baud rate is in force when you press ENTER or H followed by ENTER to the power up or reset prompt question:

```
Cass?
```

The 500-baud speed is selected by

```
Cass? L
```

See FHBF.

FCER
FC Error

Function Call error.

General catch-all for illegal parameter in a function, as in

```
100 A=SQR(-234.56)        'i (j?) can't do this!
110 A=II(2,-8)            'hyperspace, anyone?
120 A=USR(100000)         'system has a lot of RAM!
```

FDER
FD Error

File Data error.

Bad file data from cassette. This error refers to data values being read from a data file on cassette. Check your PRINT # statements to see that data has been written in the proper sequence to correspond to the INPUT # statements. Also refer to CTLC for cassette tape loading hints.

FHBF
500-Baud or 1500-Baud for Model III Cassette, BASIC, Selecting

On power up for a non-Disk system, the system asks

```
Cass?
```

If you press ENTER or H followed by ENTER, 1500-baud will be selected. If you enter L followed by ENTER, 500-baud will be selected. The 500-baud rate is about a 50 characters per second rate, while the 1500-baud rate is about 150 characters per second.

To change the cassette speed under BASIC program control, use

```
100 POKE 16913,0
```

to select 500-baud and

```
100 POKE 16913,1
```

to select 1500-baud.

Another way to change the cassette speed under BASIC program control: In this method, a USR call is made to the "prompt" that sets the baud rate. Calling ROM location 12354 (**3042H**) initiates the prompt message

```
Cass?
```

The user can now reply with the baud rate response, at which time a return is made to BASIC (or to another assembly language routine). Sample for Model III, non-disk BASIC:

```
100 POKE 16526,66      'ls byte of address
110 POKE 16527,48      'ms byte of address
120 A=USR(0)           'print screen contents
130 ...                'back here
```

Sample for TRSDOS/LDOS Disk BASIC:

```
100 DEFUSR0=12354      'define address
110 A=USR0(0)          'print screen contents
120 ...                'back here
```

F

FHOF
Failures, How Often?

How often will your computer system fail? Expect one or two service calls in the first six months. Once you get past the critical "burn-in" period of several dozen hours, you'll be in pretty good shape. Your disk drives may need realignment every six months or so if used often. Your printer may need some mechanical adjustments once a year.

Are there lemons, computers made on a Friday or a Monday (after a long weekend in The Cattleman or the Taipai Bar?). You bet. Half a dozen service calls in six months are too many, if there are legitimate hardware (and not operator) problems. Keep a record of the service calls, dates, and corrective action. Don't be afraid to demand a new piece of equipment if the same errors reappear *after* the warranty period that appeared *within the warranty period. You bought a computer to help you, not to hinder you!*

FHSF
Files, How System Finds

When you specify a disk file name (see FNMH), the Operating System will generally search all disk drives for that file name. It'll start at drive 0, and then go on to drives 1, 2 and 3, if you have that many. You don't really need the drive spec for disk reads, therefore, as the system will search all drives anyway.

FNFW
File Not Found When the Directory Says It's There!

Some software assumes an extension (see FNMH). The Series I Editor/Assembler, for example, looks for the extension /SRC if you try to load

```
*L NAME
```

Try the extension shown in the directory, or in some cases (as in LDOS), use a slash after the file name so that the software will not tack on its "default" extension

```
L  ARTICLE/
```

(loads ARTICLE rather than ARTICLE/SCR in LSCRIPT, the LDOS version of Scripsit).

FNMH
File Names, How to Use

TRSDOS and LDOS on the Models I, II and III, all use the same file name format. The Color DOS for the Color Computer is the same except for "password." The simplest format is:

```
FILENAME
```

where "FILENAME" is a 1 to 8-character file name. The first character of this name must be an alphabetic character, while the remaining characters may be alphabetic or numeric. Typical names might be A, ACCNTS, SPACEWAR or H001, any 8 characters you wish. (The TRS-80s *have* been programmed to reject profanity, however — such file names as "APPLE" and "IBM" are not accepted).

You can add an "extension" to this name if you want. An extension has the format:

```
FILENAME/EXT
```

where EXT is one to 3-characters, starting with an alphabetic character. Typical names with extensions might be ACCNTS/FEB or H123/BAS. There are certain "default" extensions that various programs "tack on" to the file name automatically, however. Editor assemblers usually append "/SRC" for "source" and "/CMD" for "command", LDOS Scripsit uses "/SCR," BASIC uses "/BAS," and so forth. Use extensions at will to help you identify certain common types of files — all files for "February," for example, might have the extension "/FEB."

Not Applicable to Color Computer: The next form of filenames adds a "password." A password is just what it implies — a secret word that only you and your designees know that must be used to utilize the file. The format of this file specification is

```
FILENAME.PASSWORD
```

or

```
FILENAME/EXT.PASSWORD
```

The password is 1 to 8 characters, starting with an alphabetic character. Typical passwords might be .A234567, SECRET, or .MINDY. When a file has a password, it is called a "protected" file, as a user must know this password to access the file.

The next, and last item in a disk file name is a "drive specification." The drive spec has the form :0, :1, :2, or :3, depending upon which disk drive you are accessing. A file name with a drive spec might be ACCNTS:0, ACCNTS/FEB:0, or ACCNTS/FEB.MINDY:0. Drive specs are most useful for disk writing to tell the system which drive to use for the file. On disk reads the system will search all disk drive directories to find a specified file and no drive spec is really needed.

FSG2
Video/Graphics Worksheet, Model II

Figure FSG2-1 is a full size graphics worksheet for the Model II. IJG and William Barden, Jr. hereby grant permission to copy it for your own use only as many times as is necessary, as long as the number of copies is kept under 250,000.

Figure FSG2-1 – *Graphics Worksheet, Model II*

(See next page).

FSGC
Video/Graphics Worksheet, Color Computer

Figure FSGC-1 is a full-size graphics worksheet for the Color Computer. IJG and William Barden, Jr. hereby grant permission to copy it for your own use only as many times as is necessary, as long as the number of copies is kept under 250,000.

Figure FSGC-1 – *Graphics Worksheet, Color Computer*

(See following pages).

FSGW
Video/Graphics Worksheet, Model I/III

Figure FSGW-1 is a full size graphics worksheet for the Model I/III. IJG and William Barden, Jr. hereby grant permission to copy it for your own use only as many times as is necessary, as long as the number of copies is kept under 250,000.

Figure FSGW-1 – *Graphics Worksheet, Model I/III*

(See following pages).

Figure FSG2-1 – *Graphics Worksheet, Model II*

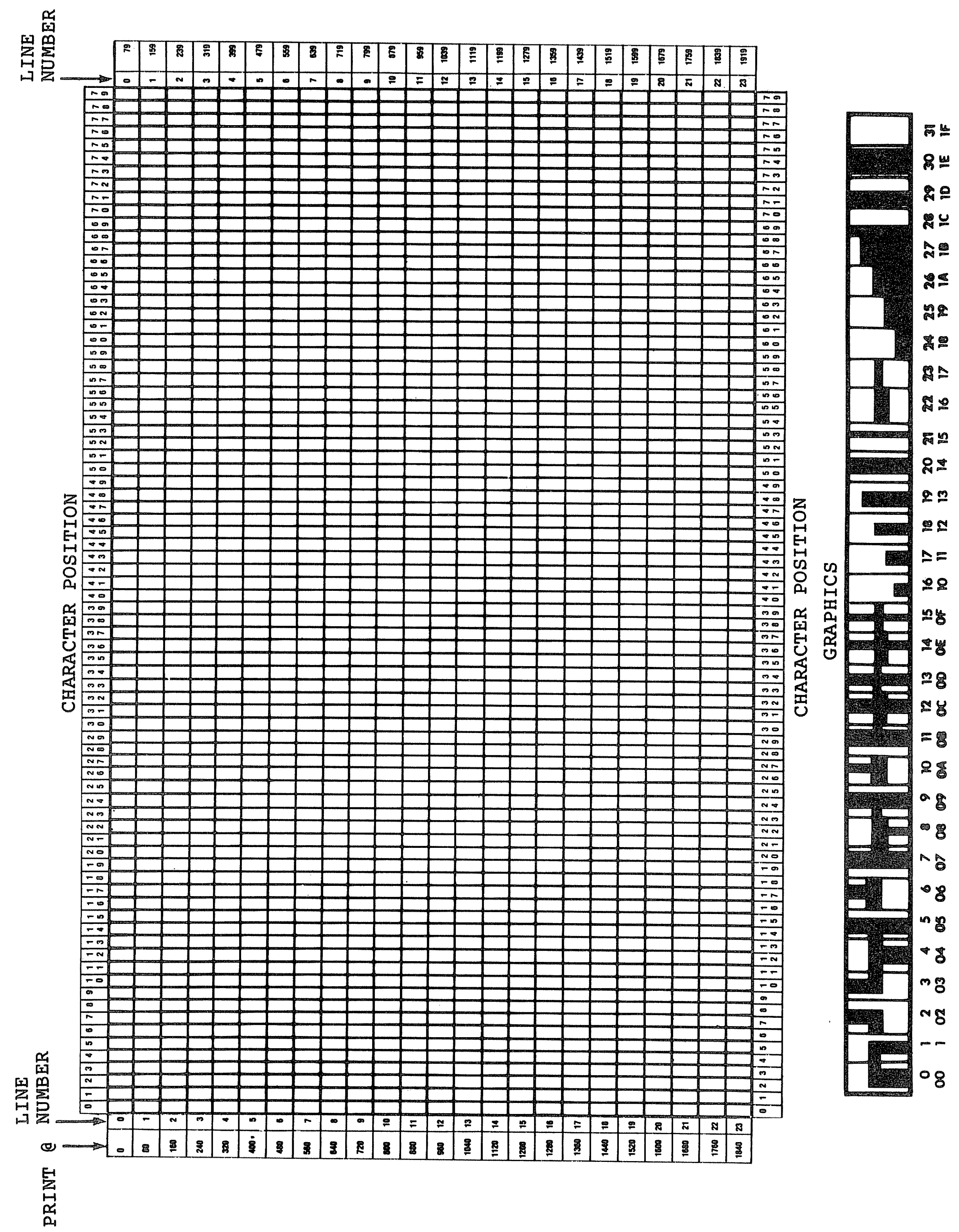


Figure FSGC-1 – *Graphics Worksheet, Color Computer*

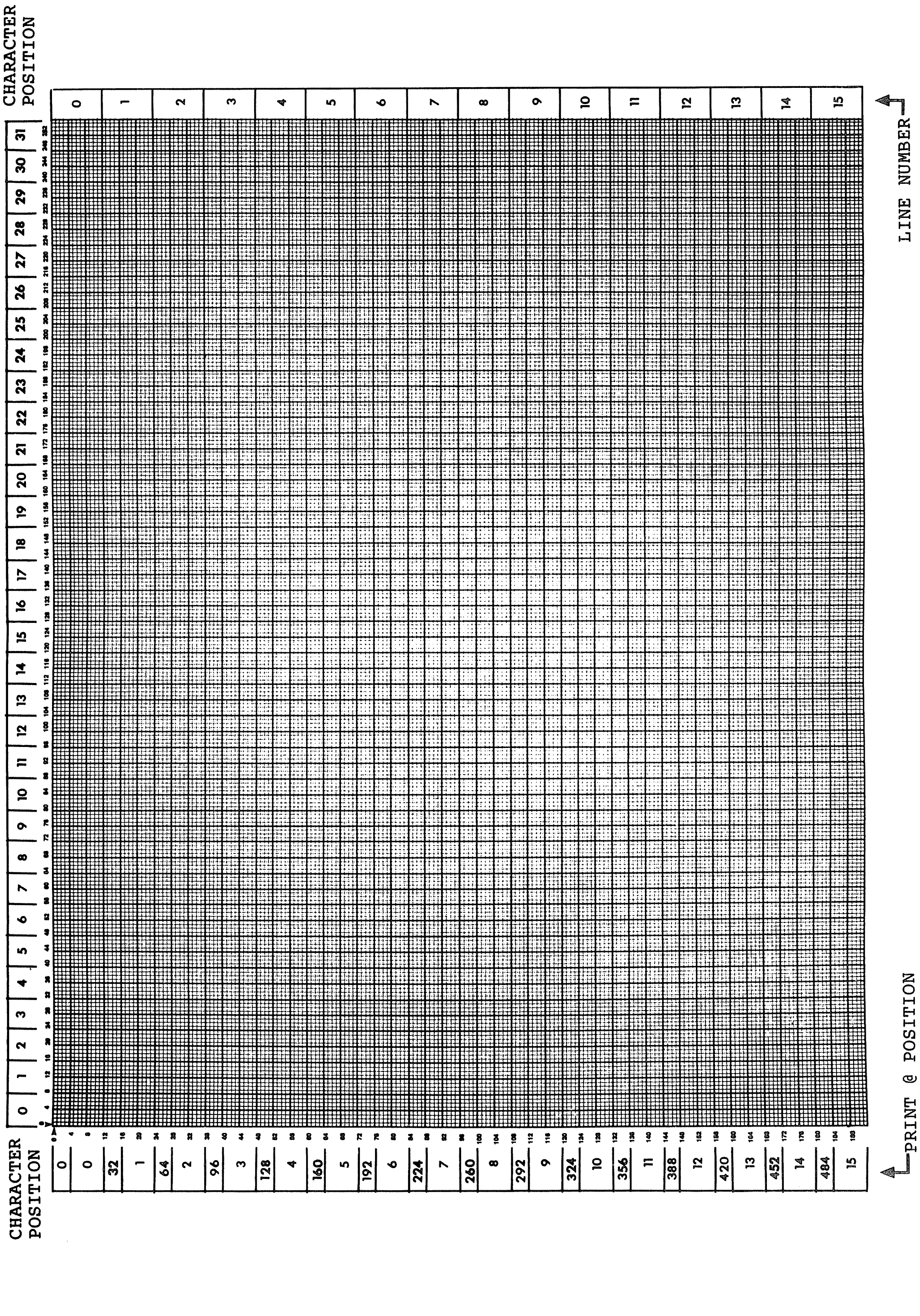
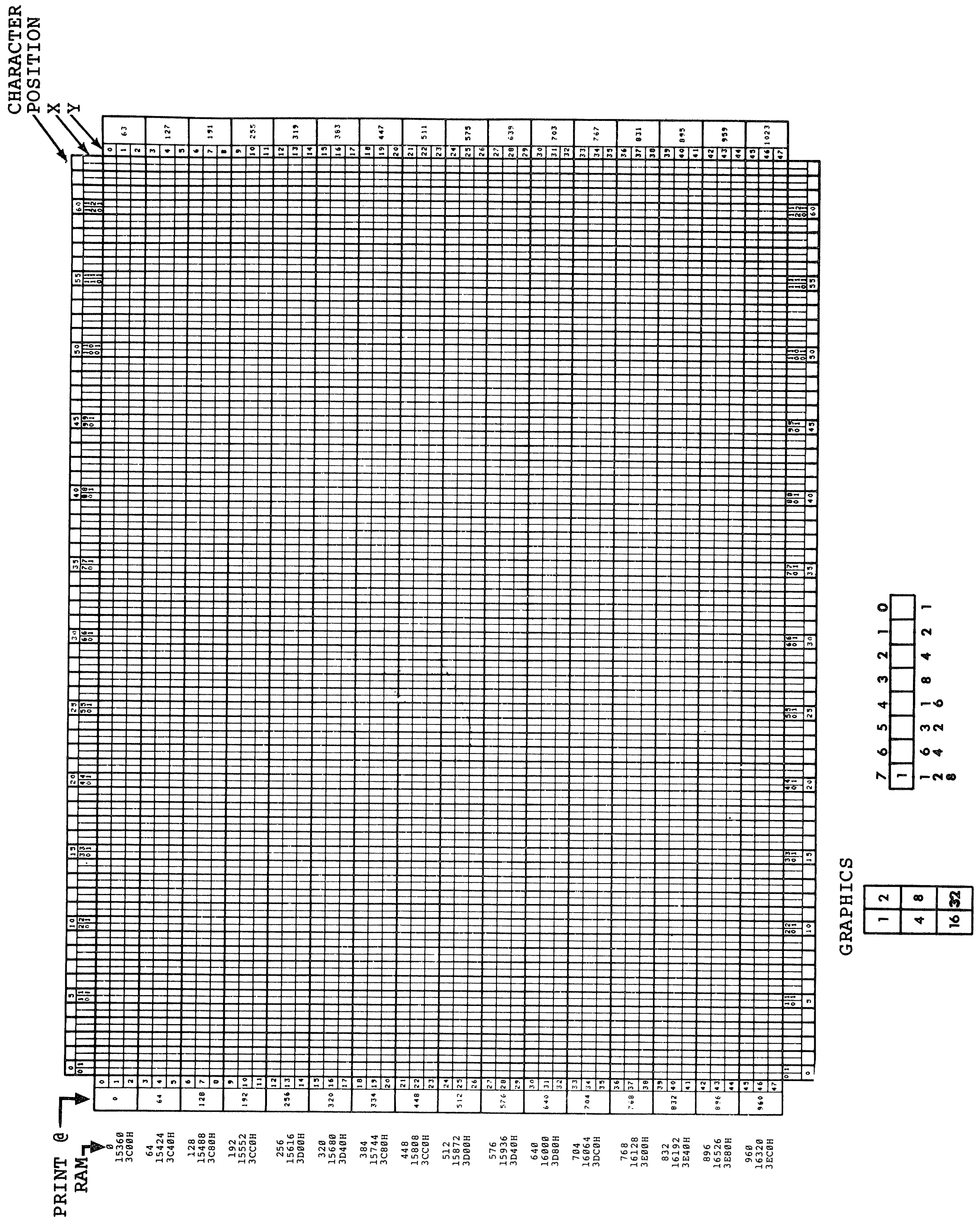


Figure FSGW-1 – *Graphics Worksheet, Model I/III*



FTLO
FOR . . . TO Loops, BASIC, Okay to Break Out? All Systems

Sure. If you have

```
100 FOR I=0 TO 1000
110 IF A=5 GOTO 200
120 A=A+ZZ(I)
130 NEXT I
```

it's perfectly fine to break out when A=5. The program will not lose control, and nothing catastrophic will happen. Variable I in this example will be set to the value it had at the "breakout" point; the remaining variables will also hold legitimate values.

See FTST for more on FOR ... TO ... STEP.

FTST
FOR ...... TO ...... STEP, BASIC, All Systems

The FOR . . . TO . . . STEP command is used together with NEXT to set up a "loop". Here's a simple case:

```
 90 A=0
100 FOR I=1 TO 100
110 A=A+I
120 NEXT I
```

The variable A is first set to 0. Line 100 is then executed, setting I to 1. Line 110 adds the value of I, now 1, to the value of A. The sum is put back into variable A.

Line 120 finds the next value of I by adding 1 to I; I is now equal to 2. If I is not the end value of 100, lines 100, 110, and 120 repeat again. This "loop" repeats until I finally reaches 100. During the process, we've got something like:

```
I   A
1   0
2   1
3   3
4   6
    .
    .
    .
100  5050  Done!
```

You'll notice that no STEP was specified. STEP determines how much is added to the "loop variable." If line 100 had stated 100 FOR I=1 TO 100 STEP 3, the increment value would have been 3, and I would have been 1, 3, 6, 9, . . .

We also could have started with any starting value and ended with any ending value, as long as the values were 0 through 32,767, or -1 through -32,768. We could have had 100 FOR I=100 TO 30000 STEP 6, for example, or 100 FOR I=10000 to 100 STEP -10. Note that the STEP must be a negative value if we are STEPping in reverse.

The values used in the FOR . . . TO . . . STEP statement can be constants, variables, or expressions. You could have 100 FOR Z = (Y*2) TO (D/100), for example.

Also, though not generally done, the start, end and STEP values may be "mixed" numbers, such as 45.67. The variable used in the loop as the "control" variable may be any variable type. Also, the end value does not have to be a "multiple" of the step value; ending at 100 and STEPping by 30 is all right, for example.

The NEXT value can be located any number of lines forward in the program.

See NFER for information on "nesting" FOR . . . TO . . . STEP loops and FTLO for "breaking out" of these loops.

notes

GAPC
GET and PUT use in BASIC Graphics, Color Computer

GET and PUT are not complicated in what they do, but how they do it. Basically, PUT takes a portion of a graphics screen and stores it into a two-dimensional array. At a later time, GET retrieves the information from the array and reconstructs the screen segment somewhere else on the screen. This can be done fairly rapidly. (Rapidly enough for moving large figures for animation). The whole process is shown in Figure GAPC-1.

GET and PUT are ideal for animation or for saving blocks of graphics which can later be called up to construct composite figures. Since the number of arrays that can be used is limited only by RAM, you can have many different graphics blocks stored and available for display. The blocks could represent characters or a set of predefined figures. A logic diagram could be drawn by using one PUT array as an AND gate, one as an OR gate, one as a NAND gate, and so forth.

One word to the wise here. Although Radio Shack documentation implies that the array must be the same size as the graphics block, eating up huge chunks of RAM, in fact, the array may be made considerably smaller. A "one by n" array in the form AR(0,N), where N is determined by the method in Figure GAPC-2, can be used to create much larger GET/PUT areas in less space. Thanks to James Garon for this one.

Figure GAPC-1 – *GET/PUT Action*

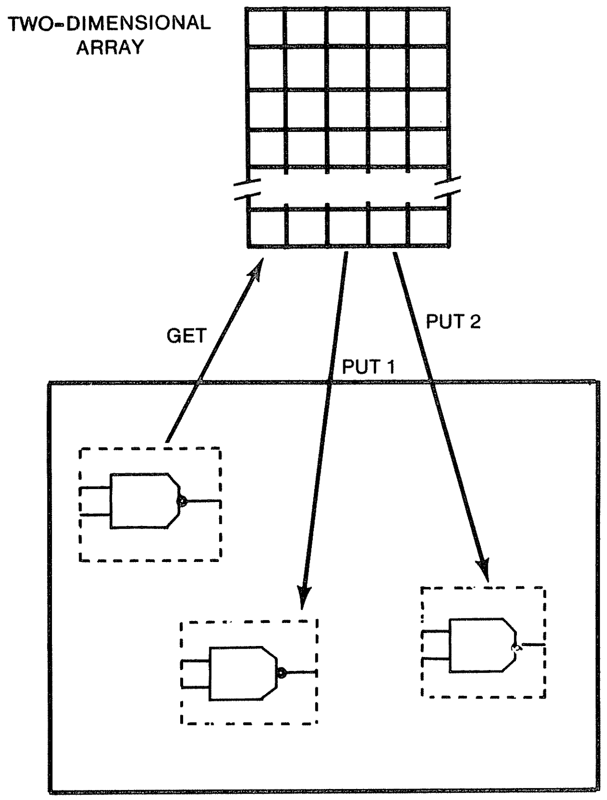


Figure GAPC-2 – *Calculating GET/PUT Array Size*

```
1ØØ PMODE 3,1
11Ø GET (42,42) - (1Ø6,1Ø6)

STEPS:

1. FIND ELEMENTS IN GET:
      1Ø6         1Ø6
      -42         -42
       64          64
       +1          +1
       65    *     65   =   4225 ELEMENTS

2. FIND DIVISOR D:
      PMODE      "G" OR NOT "G"    DIVISOR
        Ø             "G"            32
      1 OR 2          "G"            16
      3 OR 4          "G"             8<-THIS
        Ø           NOT "G"          32  EXAMPLE
      1 OR 2        NOT "G"           8
      3 OR 4        NOT "G"          16

3. DIVIDE # OF ELEMENTS BY DIVISOR D:
      4225/8 = 528 1/8

4. ROUND UP RESULT:
      528 1/8 ROUNDED UP = 529

5. FIND DIM BY DIVIDING RESULT BY 5, ROUNDING UP
   (5 IS # OF BYTES IN ARRAY ENTRY):
   529/5 = 1Ø5 4/5, ROUNDED UP IS 1Ø6

6. ESTABLISH ARRAY:
               DIM AR(Ø,1Ø6)
                     ↗    ↖
           THIS            THIS DIMENSION
         DIMENSION         FROM RESULT
          ALWAYS
            Ø
```

GC13
Graphics Characters, Model I/III

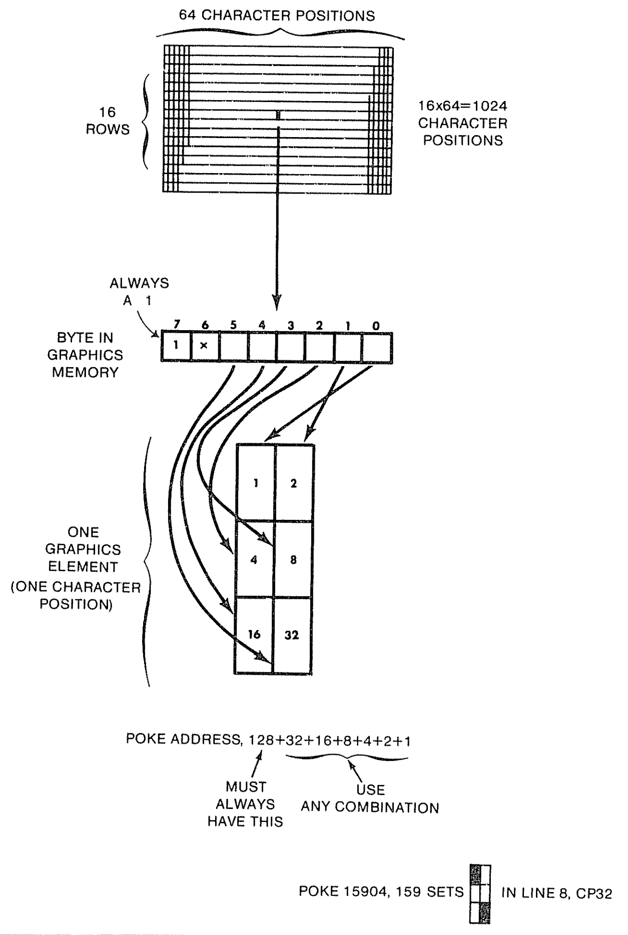
There are 1024 character positions on the screen of the Models I and III, divided up into 16 lines of 64 characters each.

Each character position is a byte in video memory (see MMM1). To find the location of a particular line and character position in video memory, multiply the line number (0-15) by 64, add the character position on the line (0-63), and add 15360. Line 8, character position 32, for example, would be located at location 8*64+32+15360 = 512+32+15360 = 15904.

You can POKE (see PPKU) any value into the video memory locations. Values corresponding to ASCII characters (see ADFW) will result in a display of text characters, values corresponding to "control codes" will cause certain display actions, or will be ignored. Values from 128 through 255 will be treated as graphics characters and will set or reset the six subdivisions of the character position. You can intermix graphics characters and text characters on a character position basis.

The graphics subdivisions of a character position are shown in Figure GC13-1. There are 6, each defined by one of 6 bits in the byte stored in the video display memory. The first bit of each byte is always a 1, as the value is 128-255 (see CFDB). The next bit is ignored. The "lower" 6 bits define the graphics character, as shown in the figure.

Figure GC13-1 – *Graphics Character Representation*

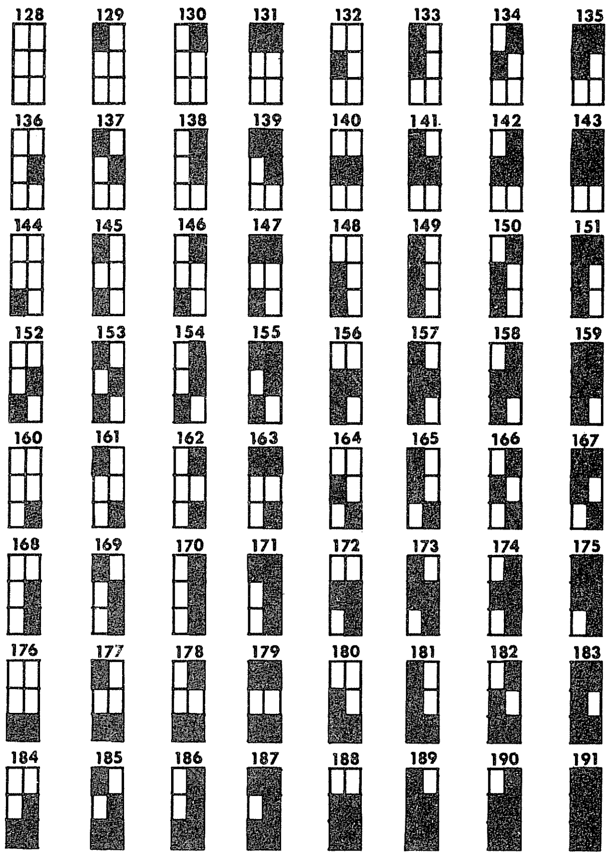


To set any graphics subdivision, set the corresponding bit to a 1; the subdivision will be white. To reset any graphics subdivision, set the corresponding bit to a 0; the subdivision will be black.

There are 64 different graphics characters that can be displayed. They and their corresponding codes are shown in Figure GC13-2.

The SET/RESET command (see SRBH) works through BASIC to set each graphics subdivision. For faster methods, see GHS1.

Figure GC13-2 – *Graphics Character Codes*



GHS1
Graphics, High-Speed, Model I/III

If you don't know where video memory is located, what a graphics character is, or if you have plenty of free time, read over GC13.

There are 3 methods for high-speed graphics. (Don't you just hate statements like that? You *know* somebody is going to say *4*!)

First, forget about SET/RESET (see SRBH). This is fine for plotting, but terrible for lines, animation, or simple figures.

Method 1: The POKE method. This method POKEs (see PPKU) graphics bytes into video memory. To use this method:

A. Determine what the configuration of the graphics characters should be, and the locations to be POKEed. You'll need to lay out a figure on a graphics worksheet (see FSGW) and translate into the graphics codes and locations.

Suppose that we have a figure of a fish. By the way, I first saw this animation by James Garon. He invented the figure and I am simply stealing it here. Are fish figures copyrightable? I'd better give him a free copy of this book The figure is shown in GHS1-1.

B. Now use BASIC (or assembly language) to POKE the graphics data. Sample:

```
100 CLS
110 DATA 130,173,180,184,188,191,191,156,180
120 DATA 160,158,135,139,143,191,191,143,135
130 FOR I=0 TO 8:READ A:POKE 15832+I,A:NEXT I
140 FOR I=0 TO 8:READ A:POKE 15896+I,A:NEXT I
```

Method 2: The String Method (Again, credit to James Garon). This method defines each row of graphics characters as a string. As strings can be PRINTed quickly, it is very fast. To use this method:

A. Determine the graphics characters, but not the locations, as in 1A above.

B. Establish a string for each row of the character. The first row would be a string of the characters 130, 173, 180, 184, 188, 191, 191, 156, and 180. A string can be made of non-text characters by using CHR$ (see CUSC). The string for the first row would be defined by

```
100 A$=CHR$(130)+CHR$(173)+CHR$(180)+...
```

C. Rapidly write out these rows at the proper screen locations by using PRINT @. Typical code would be

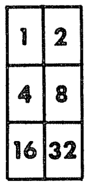
```
100 CLS:CLEAR 1000
110 A$=CHR$(130)+CHR$(173)+CHR$(180)+CHR$(184)+
    CHR$(188)+CHR$(191)+CHR$(191)+CHR$(156)+CHR$(180)
120 B$=CHR$(160)+CHR$(158)+CHR$(135)+CHR$(139)+
    CHR$(143)+CHR$(191)+CHR$(191)+CHR$(143)+CHR$(135)
130 PRINT @ 530,A$
140 PRINT @ 594,B$
150 GOTO 150
```

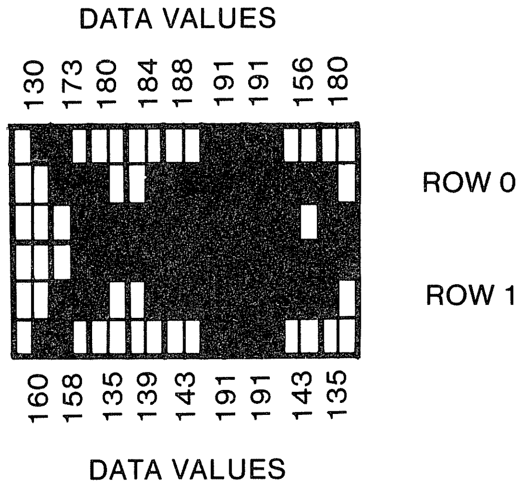
Method 3. This is a variation of Method 2 that uses cursor control characters. The cursor control characters can move the cursor to any position, and all of the rows of the figure can be defined by a single string, provided it is less than 255 characters. Typical code is:

```
100 CLS:CLEAR 1000
110 A$=CHR$(130)+CHR$(173)+CHR$(180)+CHR$(184)+
    CHR$(188)+CHR$(191)+CHR$(191)+CHR$(156)+
    CHR$(180)+CHR$(26)+STRING$(9,24)+CHR$(160)+
    CHR$(158)+CHR$(135)+CHR$(139)+CHR$(143)+CHR$(191)+
    CHR$(191)+CHR$(143)+ CHR$(135)
130 PRINT @ 530,A$
150 GOTO 150
```

There are other methods (*10!*), but these are fast and easy to use.

Figure GHS1-1 – *Graphics Figures*





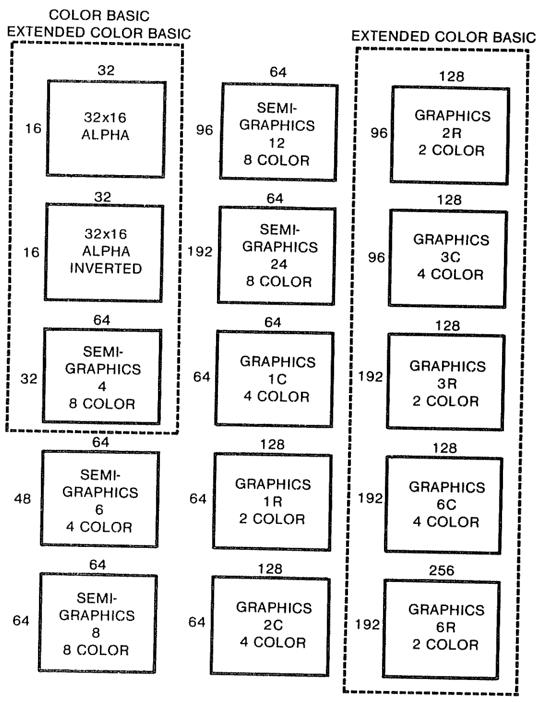
GMIC
Graphics Modes in the Color Computer

There are 15 separate and distinct graphics modes available on the Color Computer. These are shown in Figure GMIC-1. Of these, only 3 are supported in Color BASIC and only 8 are supported in Extended Color BASIC. The number of graphics modes is determined by the LSI chip used to implement text and graphics, the Motorola MC6847 Video Display Generator Chip, aka the "VDG."

If you are interested in getting down to the IC level, I'd advise buying a copy of the TRS-80 Color Computer Service Manual (RS 26-3001/3002) and the Motorola "Data Library" manual, which describes many of the Color Computer LSI chips.

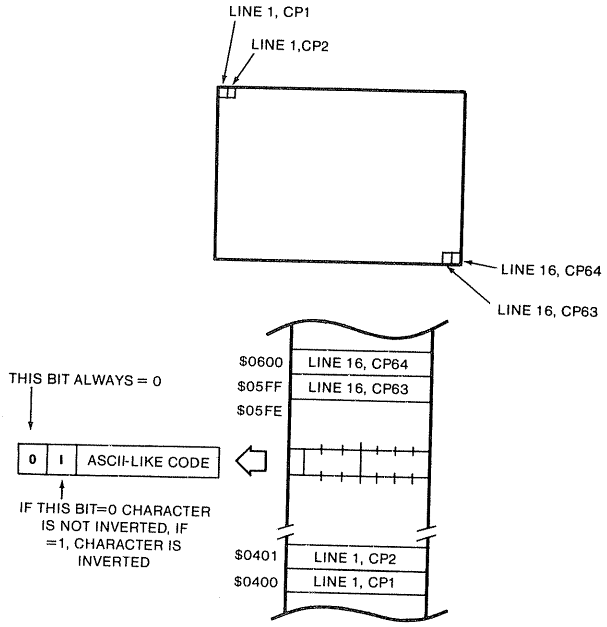
Graphics Modes in Color BASIC: Color BASIC uses only the text screen in either the alphanumeric mode or the VDG "Semigraphic 4" mode.

Figure GMIC-1 – *Color Computer Graphics Modes*



Alphanumeric Mode: In the alphanumeric mode, each byte of the text screen RAM represents a single character position in the straightforward mapping of Figure GMIC-2. Character coding is roughly ASCII (see ADFW). A single bit determines "inverted" or "non-inverted" display of text.

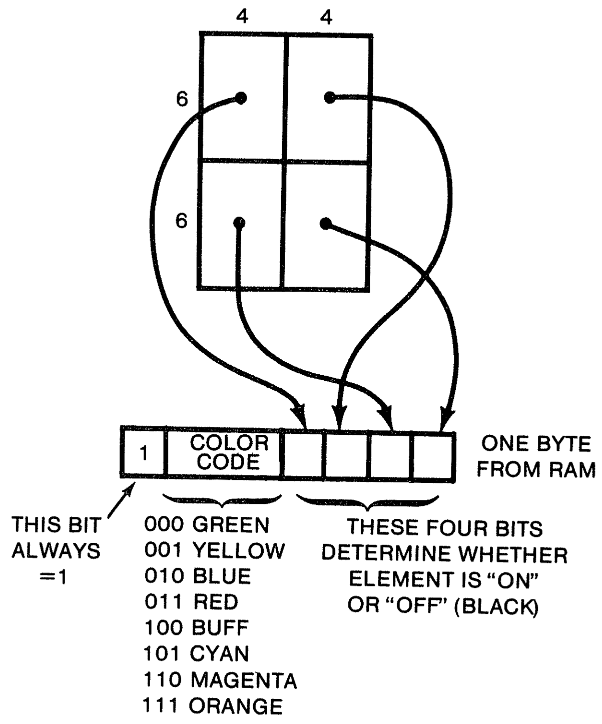
Figure GMIC-2 – *Alphanumeric Mode Mapping*



Semigraphics 4 Mode: The Semigraphics 4 mode is similar to Model I/III graphics. It allows 64- by 32-element graphics.

In Semigraphics 4, each character position of the 512 character positions of the text screen represents four elements of graphics as shown in Figure GMIC-3. The most significant bit of the character position byte is set to indicate graphics, the next three bits contain a color code of 0 through 7, and the next four bits represent the on/off status of the four pixels.

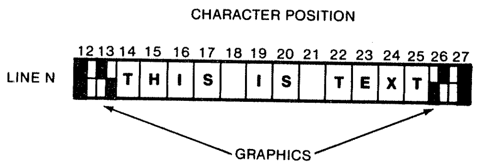
Figure GMIC-3 – *Semigraphics 4 Mode Character Position Mapping*



Color Codes: A word about color codes here. In all the following discussion the color codes will be: green, yellow, blue, red, buff, cyan, magenta, orange. (If you can learn to differentiate between these shades, a fascinating career in fashion design awaits you). These codes are 0 through 7 in "hardware" (as in the graphics byte) or 1 through 8 in a BASIC command. The BASIC color code of 0 represents black, or no color on the screen.

Typical Alphanumeric and Semigraphics 4 Displays: Semigraphics 4 and alphanumeric modes can be intermixed on the text screen, that is, text data can be displayed next to graphics data as shown in Figure GMIC-4. If you wish, an entire block can represent graphics data.

Figure GMIC-4 – *Intermixing of Text and Graphics with Semigraphics 4 Mode*



The Semigraphics 4 mode is really somewhat misleading. It's true that with a pixel being "off" (black) or "on" (color specified in byte), you can get a 64- by 32-element resolution in two colors. However, all four pixels of each character position must have the same color. This means if you intermix colors, you can't have dissimilar colors adjacent to each other unless the new color starts at a "character position boundary."

Color BASIC allows a SET, RESET, and POINT that does just what you'd expect. You may set, reset, or test each of 64 by 32 pixels with the above color limitation. A clear screen (CLS) is also provided; the screen may be cleared to any color.

An important point — if you own a Color Computer and do not have Extended Color BASIC, you should be tried by a jury of your peers, found guilty, and be given 20 years of hard labor trying to implement "Alien Tourist Invaders" on the Color Computer with only Color BASIC. Don't even waste your time with Color BASIC's SET, RESET, and POINT! Force a smile at that Radio Shack store manager and order Extended Color BASIC.

Extended Color BASIC

Extended Color BASIC supports 5 of the higher - resolution graphics modes, as shown in Figure GMIC-5. (Three lower "true" graphics modes remain unsupported, and the 3 Semigraphics Modes are still not supported.)

All of these graphics modes use the same mapping in graphics pages of RAM memory. If the mode is a two-color mode, the color code for each pixel is held in one bit, and a byte represents eight elements. If the mode is a four-color mode, the color code for each pixel is held in two bits, and a byte represents four pixels as shown in Figure GMIC-6.

Figure GMIC-6 – *Extended Color BASIC Graphics Mapping*

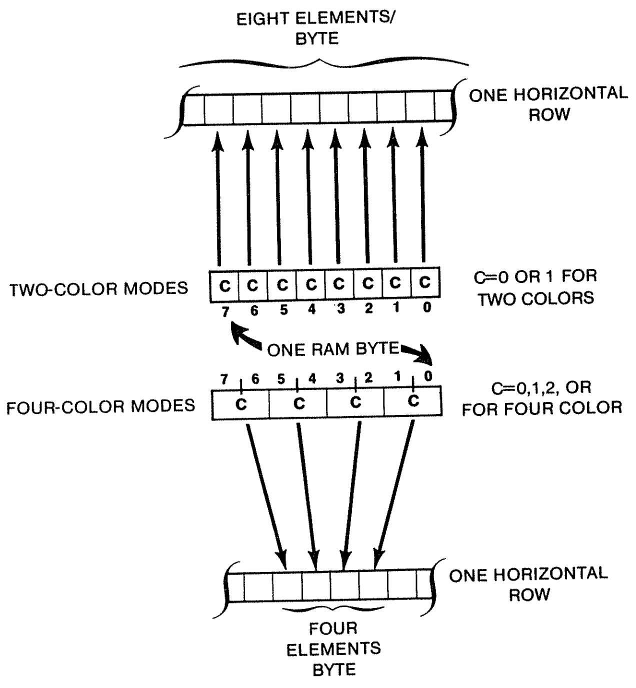
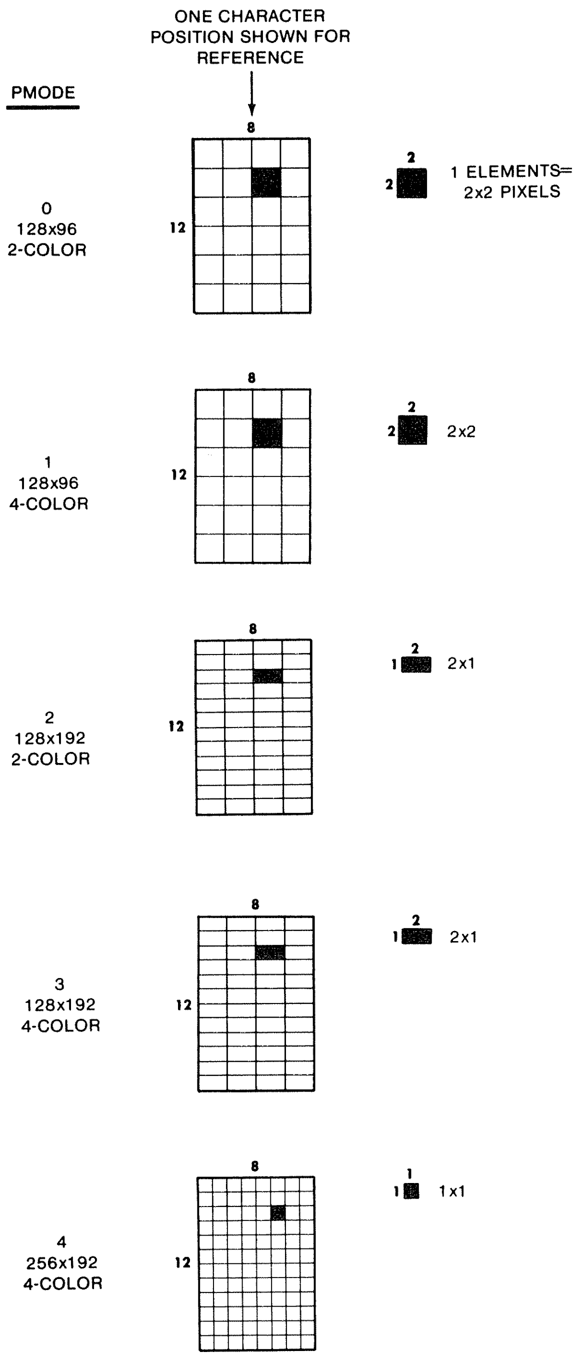


Figure GMIC-5 – *Pixel Addressing in Extended Color BASIC*



GMIC

GMIC

G

Color Sets

Two sets of two or four colors can be selected by the SCREEN command, as shown below. Use

```
SCREEN s,m
```

where s is a 0 for text screen display or a 1 for graphics screen display, and m is a 0 or 1 for selecting one of the 2 color sets. The border in Extended Color BASIC is either green (color set 0) or buff(color set 1).

Table GMIC-7 – *Color Sets for Extended Color BASIC*

TWO-COLOR COLORS

C	COLOR SET	COLOR	BORDER
Ø	Ø	BLACK	
			GREEN
		GREEN	
Ø	1	BLACK	
			BUFF
1	1	BUFF	

FOUR-COLOR COLORS

C	COLOR SET	COLOR	BORDER
Ø	Ø	GREEN	
1	Ø	YELLOW	
			GREEN
2	Ø	BLUE	
3	Ø	RED	
Ø	1	BUFF	
1	1	CYAN	
			BUFF
2	1	MAGENTA	
3	1	ORANGE	

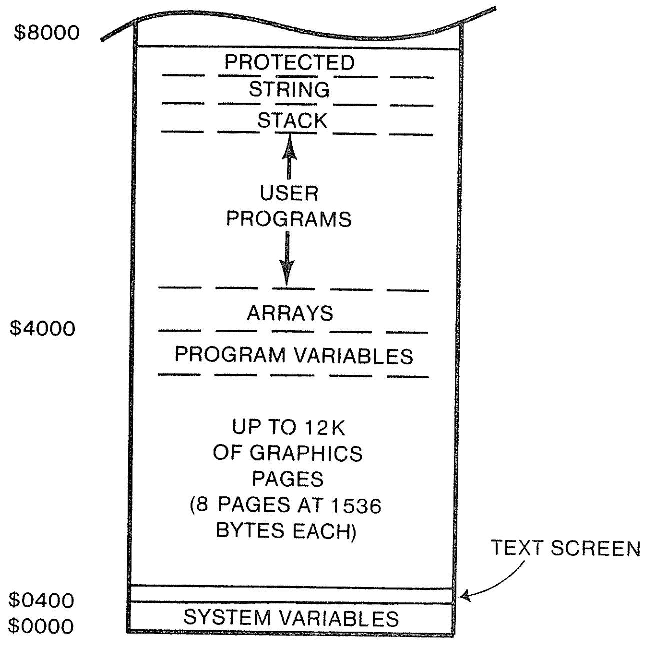
GMPC
Graphics Mapping in the Color Computer

Refer to MMCC for the general memory map of the Color Computer.

RAM occupies the first 32K, while ROM is in the upper 32K. Standard Color BASIC is 8K bytes and is in the **$A000** area ("$" stands for hexadecimal in the Color Computer). The Extended Color BASIC is an additional 8K of ROM in the **$8000** area.

The upper 16K is dedicated to cartridge ROM and I/O addresses. The I/O addresses are addresses of Color Computer "PIA" chips and hardware "vectors."

Figure GMPC-1 – *RAM Storage in the Color Computer*



The 32K of RAM is used for storage of system variables, a "text" screen, from 1 to 8 graphics screens, program variables, arrays, user programs, stack, string working storage and an optional protected machine language area (in that order) from bottom of memory to top. Now see Figure GMPC-1.

Text Screen: The area from **$400** to **$5FF** (1024 through 1535 decimal) is dedicated to the "text" screen. The text screen corresponds to text "video display memory" and is used to hold text as entered from the keyboard, text during BASIC execution, and general system text.

The text screen holds 16 lines of 32 characters, for a grand total of 512 characters, or 512 bytes.

In fact, although this text screen is mapped at the **$400** area, it can be remapped dynamically by redefining the text screen area. This is done by sending a new address to one of the dedicated I/O addresses. See CDOC.

Graphics Pages: There are from 1 to 8 graphics "pages" above the text screen. The first page starts at **$0600** and extends to **$0BFF** (1536 bytes), the next starts at **$0C00**, and so forth. You may allocate from 1 to 8 graphics pages (but not 0), depending upon the graphics modes you'll be using. Use

```
PCLEAR n
```

to select 1 to 8 pages. Any remaining space is released to the common pool of memory and used for variable storage, arrays, etc. A graphics display is made up of from 1 to 4 graphics pages depending upon the graphics mode.

The graphics pages are separate from the text pages. Under program control, you can switch from text page to graphics page (using the SCREEN command SUCG) and display either text data or graphics.

It's important to note that you can process a graphics page while displaying a text page and vice versa. In other words, you could be displaying BASIC text while loading up a graphics page with graphics art. Similarly, you could be displaying a graphics picture on one page or set of pages while loading up another graphics page or set of pages with a second picture.

Each graphics page is 1536 bytes for an excellent reason — the lowest resolution graphics mode supported in BASIC is 128 by 96 elements with two colors in each element. As two colors can be encoded in one bit, the number of bits required for this resolution is 128*96 or 12,288 bits or 1536 bytes.

The highest resolution graphics mode is 256 by 192 in two colors, requiring 256*192=49,152 bits or 6144 bytes or 4 graphics pages. You can see that you've got to plan ahead in allocating graphics pages (by the PCLEAR command SUCG) — too many and you're wasting RAM, too few and you won't have enough memory for your high-resolution graphics displays.

GPAR
Graphics Architecture, Color Computer

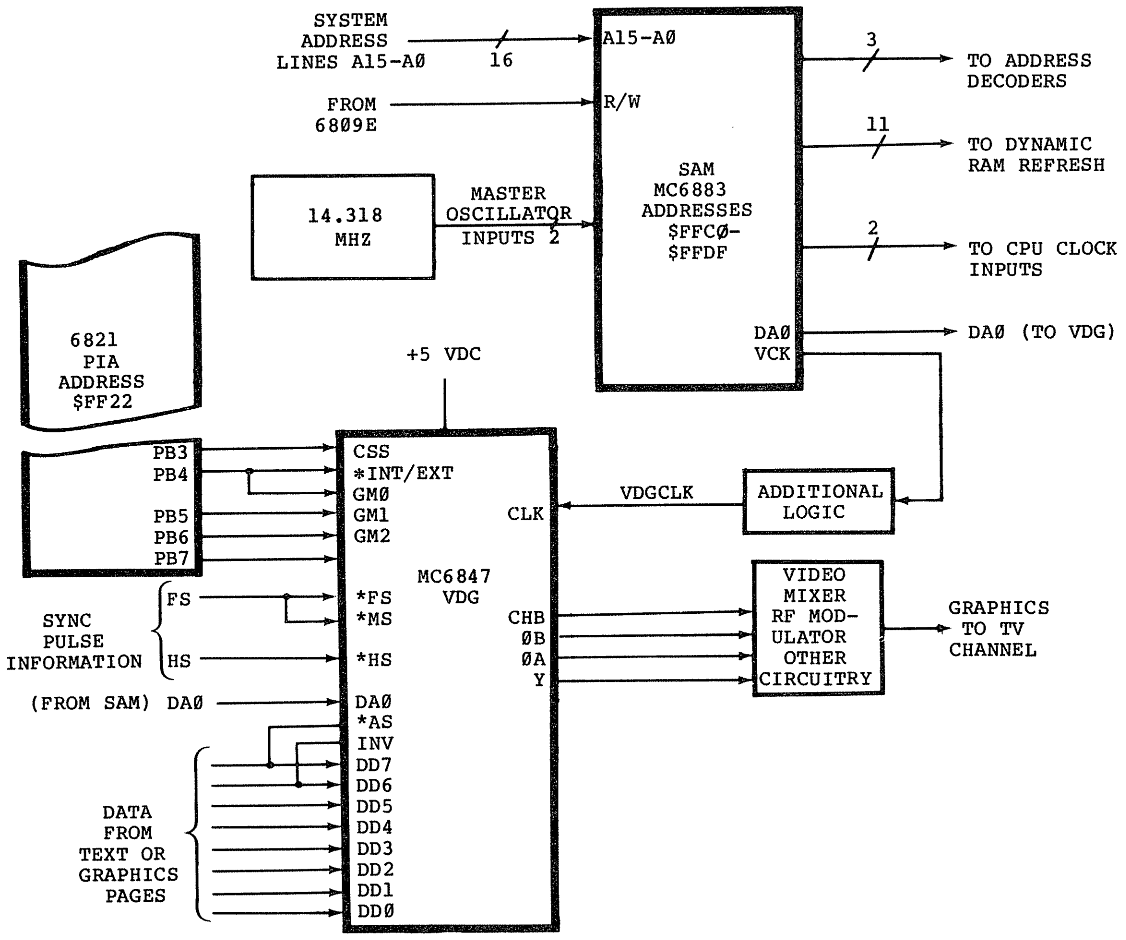
Figure GPAR-1 is a simplified diagram of the Color Computer graphics logic. You'll find a complete diagram in the Color Computer Service manual.

The heart of the text and graphics generation is the Video Display Generator Chip, the Motorola MC6847. This chip inputs RAM memory data from either the text or graphics page, converts it to a dot pattern of text or graphics, and outputs it to a color television via a video mixer chip and RF modulator.

The VDG is not addressed directly, but receives its control signals from a PIA and a Synchronous Address Multiplexer (SAM) chip. These control signals determine the graphics mode. Data from RAM is received from 8 additional inputs.

The SAM synchronizes all Color Computer system timing. It takes the master clock frequency and generates timing signals that control video display and refresh of the dynamic RAMs. It also acts as an address decoder that selects and enables ROM, RAM, or PIAs.

Figure GPAR-1 – *Simplified Block Diagram of Graphics Logic*



The SAM chip is set by a somewhat unique scheme, as shown in Figure GPAR-2. Addressing locations **$FFC0** through **$FFDF** control SAM bits for map, type, memory size, CPU rate, page 1, display offset, and VDG mode. The only two "fields" that will concern us for graphics are display offset and VDG mode.

Even-numbered addresses for the SAM reset a SAM bit; odd-numbered addresses set a SAM bit. No actual 0 or 1 is passed over data lines; the addressing action itself does the set or reset. The addressing can be done by a POKE (say, POKE **$FFC0**,0 to reset V0) or by an assembly-language store (STA $FFC0).

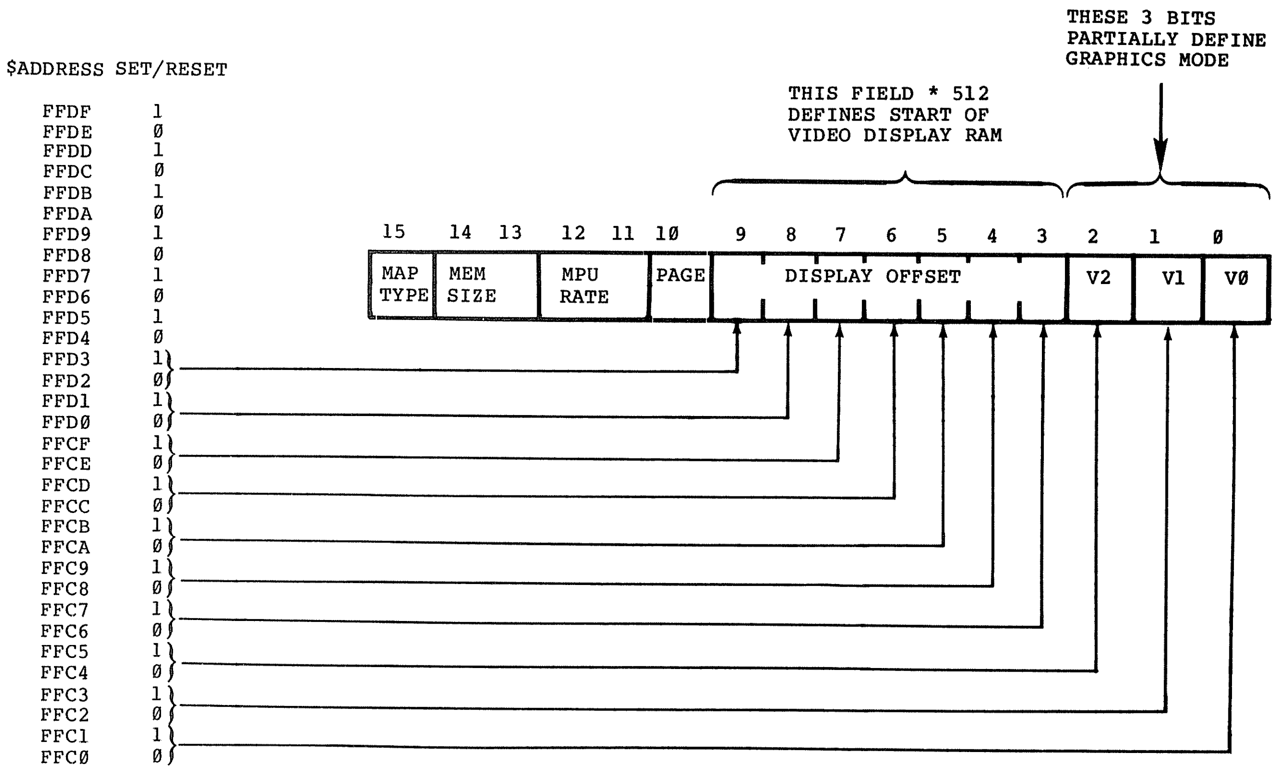
PIAs, or Peripheral Interface Adapters, are general-purpose I/O devices that interface between the 6809E CPU and system internal devices or the outside world, such as cassette inputs and outputs. The PIAs used in the Color Computer have two sets of 8 lines that may be programmed to be either inputs or outputs. Four additional lines control special functions. The PIA used with color graphics is a PIA with an address of **$FF22.**

To output data to the graphics PIA at **$FF22,** a POKE or assembly language instruction must be executed. The normal sequence that must be followed, however, is to leave the bits that are used for other functions in the PIA undisturbed. This can be done in BASIC by

```
1ØØ A=(PEEK(&HFF22) AND 7)
11Ø POKE $FF22, (X*8 OR A)
```

where X is a value of 0 through 31 that corresponds to a VDG control value.

Figure GPAR-2 – *SAM Control Bits*



HASA
Amount of Storage in a BASIC Array, How to Find

Easy. Take the amount of storage in a single element of an array and multiply by the number of elements in an array. Aha! You don't know the amount of storage occupied by a single element! Guard! Remove this poor excuse for a computer user . . .

In the Color Computer: Every numeric variable uses 5 bytes, and therefore each numeric array element uses 5 bytes. String arrays are initially allocated 0 bytes for each element after the DIM statement and thereafter are variable, depending upon how many characters the strings use. Each element in the string array has a 5-byte description block.

In the Models I, II, and III: Integer variables use 2 bytes each, and therefore each integer array element uses 2 bytes. Single-precision variables use 4 bytes and therefore . . . Double-precision variables use 8 bytes and therefore . . . String variables are assigned 0 bytes when first defined by the DIM and thereafter are variable depending upon how many characters the strings use. Each element in a string array has a 3-byte description block.

All Systems: To find the total array storage, multiply each dimension of the array plus one together and multiply this value times the number of bytes per element. The total will be close to the total number of bytes used for the array. For example, a three-dimensional Model III single-precision array defined by DIM ZZ(3,7,3) would have about (3+1) times (7+1) times (3+1) times 4 = 4*8*4*4 = 512 bytes of storage.

Now, about that "about." Besides the body of the array, there's an array variable "header" that describes the array dimensions, holds the array name, and so forth. This occupies 8 or more bytes depending upon the dimensions of the array as follows: One D: 8 bytes, Two D: 10 bytes, Three D: 12 bytes, Four D: 14 bytes. For string arrays only, also add 3 (I/II/III) or 5 (CC) bytes for each element of the array for string descriptor blocks. Confused?

Numeric Arrays: (# elements)*(# bytes per element) + (number of dimensions) * 2 + 6.

Model I/II/III string arrays: (# elements) * (# bytes per string) + (# elements*3) + (# of dimensions*2) + 6.

CC string arrays: (# elements * # bytes per string) + (# elements*5) + (# of dimensions*2) + 6.

Maybe you could give up computers as a hobby and go into collecting balls of string . . .

HMF1
How Many Files, Model I/III

After Disk BASIC has been loaded, it will ask

```
HOW MANY FILES?
```

If you press enter, Disk BASIC will "default" to three file buffers. You can specify any number of files by replying to the prompt with the number of files required.

I'm glad you asked. The number required is a function of your BASIC program and the amount of disk activity. One file buffer is required for every file OPENed at the same time. If you have no disk activity (if you don't know what I'm talking about here), simply press ENTER. If you do have disk activity, allocate one buffer for every OPENed file if they will all be opened at the same time, or for the maximum number OPENed at any time.

Enter a "V" after the file number (Model III only) if the file record lengths will be variable and set when the file is OPENed. Otherwise, the file logical record length will be set to 256 bytes.

```
HOW MANY FILES? 4V
```

allocates 4 file buffers and specifies variable record lengths.

HNIB
Hexadecimal Notation in BASIC, Some Systems

Hexadecimal notation is a shorthand form of binary. See procedures CFDH and CBBH for information on how to convert between decimal and hexadecimal and between binary and hexadecimal.

Model I/III Disk BASIC, Model II BASIC, and Color Computer Extended Color and Disk BASIC allow hexadecimal notation. Typically, you'd use this form of notation in specifying memory addresses.

Model I, II, and III: Use the prefix "&H". A value of &H8000 is hexadecimal **8000** or decimal 32,768. The value &H34 is hexadecimal **34** or decimal 52.

Color Computer: Use the prefix "&H". A value of &H1000 is hexadecimal **1000** or decimal 4096.

Unfortunately, you cannot use the prefix for hexadecimal on many BASIC commands, such as in DATA (I/III) statements where they might be most valuable. Why? What is Truth? What is Beauty?

HTDS
Displaying at any Sceen Location in BASIC, All Systems

If in BASIC, use the PRINT AT (Model I/III Level I) or PRINT @ (other systems) command. The format of these commands is "PRINT AT XXXX,item list" or "PRINT @ XXXX,item list" where XXXX is a value of 0 through 1023 for the Model I or III, 0 through 1919 for the Model II, or 0 through 511 for the Color Computer.

The "item list" is a PRINT list similar to that used with any print command — string "literals" such as "NAME=" and variables.

The location at which the display will take place will be determined by the value used in the PRINT AT or PRINT @ command. For the Models I and III, this value divided by 64 will give a line number equal to the quotient and a remainder equal to the character position on the line. PRINTing @ 1000 (1000/64=15, remainder 40) would cause a display at line 15 (the 16th line counting from line 0) and character position 40 (the 41st character position counting from character position 0). Divide by 80 for the Model II and 32 for the Color Computer.

Remember that in the Model I and III, lines start at 0, 64, 128 . . . and are 64 characters long, that in the Model II, lines start at 0, 80, 160 . . . and are 80 characters long, and that in the Color Computer, lines start at 0, 32, 64 . . . and are 32 characters long.

A typical PRINT @ might be

```
100 PRINT @ 8*64+32,"VALUE IS:";V,"DATE IS:";D$
```

This example is for a Model I or III. Note that we defined line 8 by the "8*64" and the character position by the 32, the middle character position (well, not quite, the middle is character position 31.5).

PRINT @ (AT) is typically used to avoid "scrolling" of the screen to give a more pleasing display.

As the screen location in the Models I, III, and Color Computer is "memory mapped", you can also print by POKEing characters into screen memory. (See MMM1 and MMCC and PPKU.)

HTGB
Back to Level II BASIC from Disk BASIC, Model I TRSDOS

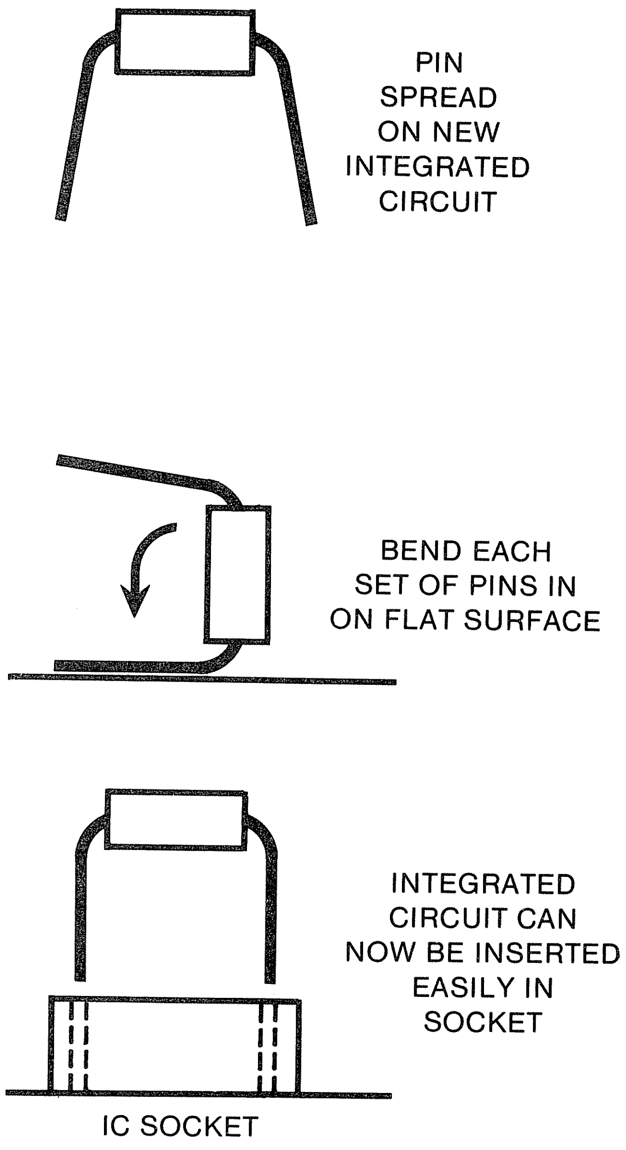
Enter BASIC2.

ICIG
Integrated Circuit, Inserting

1. Read the description of integrated circuits in LDHT if you are not familiar with them.

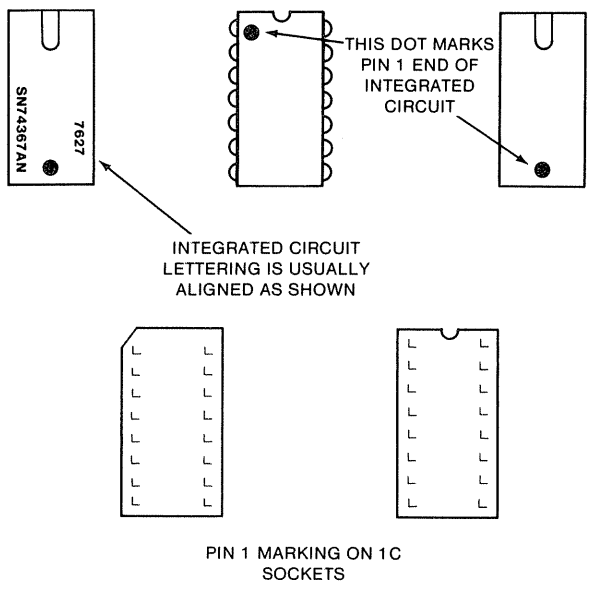
2. If this is a new part, the pins must be bent slightly in to help insertion, as shown in Figure ICIG-1.

Figure ICIG-1 – *IC Preparation for Insertion*


PIN SPREAD ON NEW INTEGRATED CIRCUIT
BEND EACH SET OF PINS IN ON FLAT SURFACE
INTEGRATED CIRCUIT CAN NOW BE INSERTED EASILY IN SOCKET
IC SOCKET

3. Position the IC so that pin 1 is in the proper position. The pin 1 end is usually marked, as shown in Figure ICIG-2. If you can't find pin 1 on the socket or the PC board holes, try the following:

Figure ICIG-2 – *Finding Pin 1*


SN74367AN
7627
THIS DOT MARKS PIN 1 END OF INTEGRATED CIRCUIT
INTEGRATED CIRCUIT LETTERING IS USUALLY ALIGNED AS SHOWN
PIN 1 MARKING ON 1C SOCKETS

A. If you have an identical board, look at an inserted component.

B. If you know the IC part number, find a logic diagram of the part, determine which pins are ground and which are positive supply voltage, and look on the PC board.

C. Grounds and voltages may be connected to a large ground or power bus that runs throughout the board, and you may be able to identify the power bus pins. If the PC board has a socket, it may be "keyed" so that the pin 1 end is marked, as shown in Figure ICIG-2.

4. Snap in the IC.

5. Verify that all leads are inside the socket holes and that no leads are bent out.

6. Double check the pin 1 location.

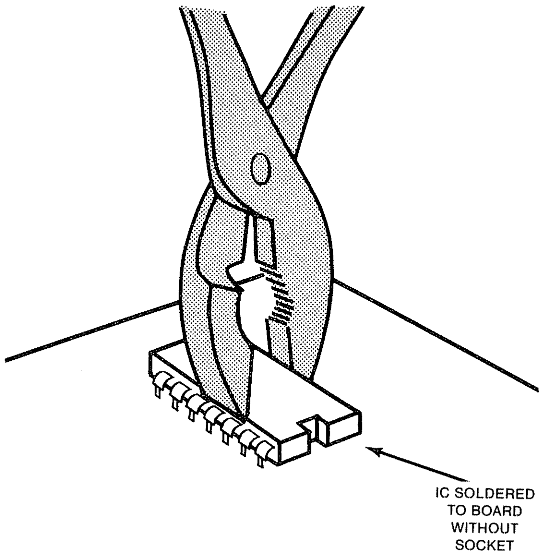
ICRS
Integrated Circuit, Removing

You have your work cut out for you.

1. If you have not soldered before, read SHTO.
2. If you don't mind destroying the chip, do the following:

A. Crush the chip with a pair of pliers, as shown in Figure ICRS-1. This sounds easier than it is. If this is not possible, try snipping off the pins with a pair of diagonal pliers.

Figure ICRS-1 – *Removing A Chip*



B. When the body of the chip has been removed, desolder the pins one by one and remove, following the desoldering procedure in SHTO. You may have to work from both sides of a printed-circuit board. Take care not to heat the printed circuit "pads" too much, as they may separate from the board. It may not be possible to perform this procedure without some pad separation.

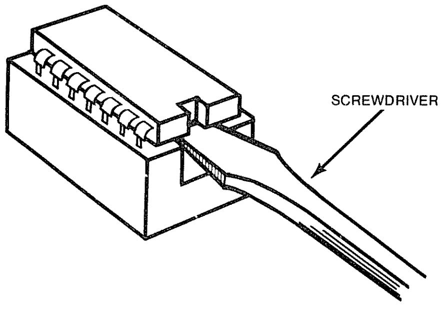
3. If you want to save the chip, you must carefully desolder the pins by the desoldering procedure in SHTO. Use a combination of the desoldering braid and desoldering tool. Wiggle the chip as each pin is heated. With luck, you'll be able to remove the entire chip. Use this procedure only for hard to find chips, otherwise destroy the bloody thing!

ICSK
Integrated Circuit, Removing From a Socket

You may use an integrated circuit puller or simply a small slot screwdriver:

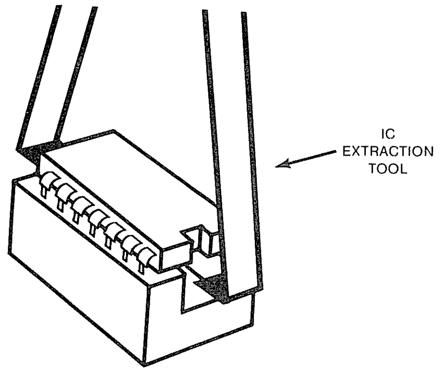
A. If using a screwdriver, slide the blade under one end, and pry up about 1/8 inch, as shown in Figure ICSK-1. Now pry up the other end. Repeat until the IC is out. Almost all ICs will come out easily.

Figure ICSK-1 – *Removing IC, Unsophisticated Method*



B. If using an IC puller, insert the blades under the ends of the IC as shown in Figure ICSK-2. Almost all ICs will come out easily.

Figure ICSK-2 – *Removing IC, Sophisticated Method*



IDER
ID Error

ID error. You have shown an invalid pass to a guard at One Tandy Center, and he has taken you down to the "interrogation room." ("What's that funny large wheel, with the shackles?!?").

Actually, "illegal indirect," an obscure term meaning that an input of a BASIC program file has occurred and a zero was found for a line number. This often occurs when a non-BASIC file is read in on a LOAD "file" or CLOAD "file." As soon as an erroneous line number (0) is found, an ID error is output. Check the file you've loaded; is it a BASIC file? See LFBA.

IDFK
INPUTting Data From the Keyboard, BASIC, All Systems

The *most rudimentary way* to input data from the keyboard is with an INPUT statement. The format of INPUT is

```
INPUT item list
```

and a typical input might be

```
1ØØ INPUT NM$, AG, NT
```

where NM$, AG, NT is an "item list" of three items. Note that the items can be any variable types and any mixture; you can have strings, integer variables, and so forth.

When the BASIC interpreter encounters the INPUT statement during program execution, it will display a question mark. You can then input the items from the keyboard. In this case, the dialogue would go something like this:

```
? WILLIAM BARDEN JR.    (ENTER key)
?? 21                   (ENTER key)
?? 4                    (ENTER key)
```

After the last item is entered, the next BASIC statement will be executed.

Another way you could have entered the data above is

```
? WILLIAM BARDEN JR.,21,4
```

In this case, the entry items were separated by commas which (in computer jargon) are "delimiters."

There must be one data item entered for every data item in the INPUT statement, and they must be in the correct order.

After the input, the variables in the list are set to the values entered. In this case NM$="WILLIAM BARDEN JR.", AG=21, and NT=4. (In case you're wondering, AG, of course, is "age" and NT is "number of TRS-80s." Didn't think you'd believe me . . . about the TRS-80s, that is).

Here are some *warnings* about using INPUT:

1. You can't INPUT a string variable with a delimiter in it, such as a comma. Use LINE INPUT instead (SDFB).

2. You cannot use commas in numeric values for the same reason.

3. You can't INPUT a numeric value greater than a numeric variable can handle. You couldn't input 57892 for an integer variable (IVHU), for example.

4. Numeric items can't include text.

There's *another form of INPUT* that you can use when you want to display a message directly before the INPUT. If we had wanted to display "ENTER NAME, AGE, # TRS-80S", we could have used

```
1ØØ INPUT "ENTER NAME, AGE, # TRS-8ØS";NM$,AG,NT
```

Note that in this case there was a semicolon directly before the item list.

See INPUT # (SDFB) to see how similar INPUT statements can input data from disk files.

INAR
Initializing Arrays, BASIC, Most Systems

Model I/III Level I does not allow arrays.

All elements of a numeric array are initially 0, and all elements of a string array are initially "null" or zero length (see SHTU).

If you would like to initialize an array to some initial value or to "zero it out" after some intermediate processing (it is zeroed out on restart), then you must store a value in each element of the array by a loop such as

```
1ØØ FOR I TO 1ØØ: FOR J=Ø TO 32
11Ø AX(I,J)=55
12Ø NEXT J: NEXT I
```

Sorry, there's no easier way.

INHU
INKEY$, How to Use, BASIC, Most Systems

(Does not apply to Model I/III Level I.)

The INKEY$ function lets you read a single character from the keyboard. Here's the catch, however. The key is present only for a fraction of a second, so the test for the key must be done rapidly. Typical code:

```
1ØØ A$=INKEY$: IF A$="" THEN GOTO 1ØØ
11Ø B$=B$+A$         'append new character
12Ø PRINT B$         'display
13Ø GOTO 1ØØ         'next key
```

The code above shows how INKEY$ works. INKEY$ returns one character. That character is a null string (zero length, indicated by "") if no key has been pressed. If a key has been pressed, the ASCII equivalent (ADFW) of the key is returned (if "A" is pressed, A$="A"). The code above checks for a "non-null" character and appends (concatenates) it to string B$. An ever larger B$ string is displayed as each key is pressed. Keep your loops "tight" for INKEY$, otherwise you will miss occasional keys; a loop that operates 10 times per second, for example, might miss keypresses.

IOHT
INP/OUT in BASIC, How to Use

Model I/III, Level II/III or Disk BASIC only: INP and OUT read and write to the system I/O ports. The system I/O ports are 256 addresses associated with an IN or OUT Z-80 instruction. Most of the addresses are not used in the Model I or III. The cassette I/O logic and RS-232-C board are assigned IN and OUT port addresses in the Model I; these devices, the disk controller, line printer, and other internal logic operations are assigned IN and OUT addresses in the Model III.

Executing an INP or OUT in BASIC is logically equivalent to executing an IN or OUT in Z-80 machine language.

The INP command reads in data from a specified I/O port.

100 A=INP(255), for example, reads 8 bits of data associated with the cassette.

The OUT command writes out 8 bits of data to a specified I/O port:

```
1ØØ OUT 255,2
```

writes out to the cassette logic.

Note that the INP requires parentheses while the OUT does not.

Generally these commands would be used only in isolated cases in the Models I and III, as BASIC is generally too slow to read or write data to cassette, disk, or RS-232-C interface. Examples of INP and OUT use are shown in LWD1 and LWD3.

IPNF
Integer Portion of a Number, Finding, BASIC, Most Systems

There are two general BASIC commands for finding the integer portion of a mixed number (fraction).

INT is used when only positive numbers are involved and returns the integer portion of a mixed number:

```
1ØØ A=INT(1Ø1/4)
```

would set variable A equal to 25, for example, instead of 25.25.

FIX is used when both positive and negative numbers are involved. (Does not apply to Model I/III, Level I or Color BASIC).

```
1ØØ A=FIX(-1Ø1/4)
```

would set variable A equal to -25. If INT were used, the result would be -26, in error.

Use INT to compute the infamous two bytes of a memory address for POKEs, PEEKs, and USR calls as follows:

```
1ØØ MS=INT(MA/256)
11Ø LS=MA-MS*256
```

This sequence will set MS, or "most significant," equal to the "high-order" byte of the memory address MA; the value in MS will be 0-255, ready for POKEing. LS will be set equal to the low-order byte of the memory address MA; the value in LS will also be 0-255.

Model II users: Use the Model II command "reverse slash" (\ ,CTRL 9) to operate on integer values between +32767 and -32768. If A were 1000 and B were 30, for example,

```
1ØØ C\B
```

would set C equal to 33 and the fractional portion of the result would be ignored.

ITEL
IF . . . THEN . . . ELSE, BASIC, All Systems

IF . . . THEN . . . ELSE can be used to "conditionally" execute a statement. If (the command is IF . . . THEN), then the statements after the THEN are executed (did you notice how this sentence itself is an IF . . . THEN?).

```
1ØØ IF A=B THEN PRINT C
```

prints variable C if variable A=B.

The "If test" that is made can use the "conditional operators" < (less than), <= (less than or equal to), = (equal to), >= (greater than or equal to), > (greater than) or <> (not equal to). The two quantities compared may be constants, variables, expressions or two string variables (A<1, B<A, C<>(I+2), A$<B$).

If the line has multiple statements, then all statements after the THEN are executed if the THEN condition is met.

```
1ØØ IF A=B THEN PRINT C: PRINT D: E=1.2
```

prints variable C and D and sets variable E equal to 1.2 if A=B. Note that if the THEN action is not taken, no other statements in the same line are executed. If A was not equal to B, C would not be printed, D would not be printed and E would not be set equal to 1.2.

(Does not apply to Model I/III, Level I).

The IF . . . THEN . . . ELSE structure works the same, except that the ELSE action is taken if the IF . . . THEN action is not true:

```
1ØØ IF A=B THEN PRINT C ELSE PRINT D: E=3.4
```

would print C if A=B else it would print D and also set variable E equal to 3.4. In this case, multiple statements following the ELSE would only be executed for the ELSE action.

Want to go crazy? You can "nest" IF... THEN... ELSE:

```
1ØØ IF A=B THEN IF A=C THEN IF A=D THEN IF A=E
    THEN B=1 ELSE B=Ø
```

would set . . . uh . . . duh . . . wait, I've got it . . . would set B equal to 1 if A=B=C=D=E or to 0 if any of the above are false. Why would you want to do this? Comes in handy more often than you would realize.

IVHU
Integer Variables in BASIC, How to Use

(Does not apply to Model I/III, Level I or Color Computer).

Integer variables require two bytes of storage and thus save RAM space. Integer variables can also be processed much more rapidly than single- or double-precision variables, speeding up execution.

Integer variables can hold values from -32,768 through +32,767. No fractions are allowed. Specify an integer variable by a suffix of "%". In the following code, A% and B% are integer variables.

```
1ØØ A%=123
11Ø B%=567
```

You can also specify all variables within a "range" as being integer by the DEFINT command. DEFINT A-G, for example, defines all variables that start with A through G as being integer; A, AA, B, FF, and GX would be integer variables in this case.

When integer variables are stored in memory, they are stored least significant byte followed by most significant byte for the Models I, II and III.

Use a suffix of "E" for single-precision numbers in scientific notation (see AOIB).

notes

JCLW
JCL, What is It? — How to Use It

JCL stands for "job control language." Picture a multimillion dollar computer system run by the CIA. Now picture Stanley Spook sitting at a video terminal, typing in his program. "Let's see, G...O...T...where's that "O"?..."

Obviously, a single user can't tie up a large system. Large systems run many "jobs" concurrently. A payroll sort is done for a short time and then a FORTRAN compile is resumed. The system takes advantage of independent input/output and other techniques to operate at maximum efficiency. The large Operating System gets its commands to do this from "JCL," which specifies which actions are to be taken for each "job."

Model II and III TRSDOS and Model I/III LDOS both have a form of JCL. In TRSDOS it's called "build-files"or "DO files;" in LDOS it's called JCL.

In both schemes, a series of commands to the DOS can be built into an ASCII file (see AFWA), one command on every line. The file can then be executed by a "DO" command which looks at each line as a DOS command.

A typical Model III JCL sequence might be

```
FORMS (WIDTH=64, LINES=16)
FREE :Ø (PRT)
FREE :1 (PRT)
LOAD ACCOUNTS
```

Unfortunately, TRSDOS really doesn't have enough flexibility to do anything significant with JCL and DO files. There are too few commands, and they are not "parameterized" enough. LDOS, on the other hand, is geared to JCL with plenty of options for commands that eliminate operator intervention, and plenty of commands, period.

To build a Do File, use the BUILD command:

```
BUILD filename
```

This command opens a file with name "filename" and then waits for your line input from the keyboard. Each line should define a DOS command in standard format. When you're done entering the lines for the file, press BREAK at the beginning of any new line. The file is then closed. The result of this operation is an ASCII file that is an "image" of your line input. LDOS automatically uses the extension /JCL on the file if none is specified; TRSDOS uses /BLD. The file in all cases is an ASCII file (see AFWA).

To execute a DO file, simply do

```
DO filename
```

The DO command will tack on the proper /BLD or /JCL extension, load the file into a buffer and then sequentially execute each line of the file as if it were typed in from the keyboard.

DO files can also be executed automatically on startup — see AUTO in AEDP.

Edit your DO or JCL files by using SCRIPSIT or LSCRIPT. They will load as normal ASCII files. You can also create DO files by SCRIPSIT or LSCRIPT, but if you do, limit the line length for TRSDOS to 80 chatacters (63 for LDOS); also, use the proper extension of /BLD or /JCL.

Model I/III LDOS only: You're allowed to append a JCL file to an existing JCL file by

```
BUILD filename (APPEND)
```

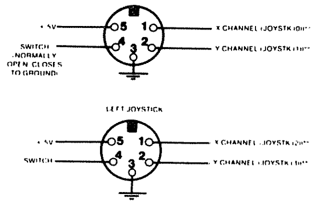
The APPENDs should be done with files created by BUILD to avoid problems with special "end of text" markers.

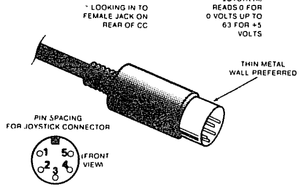
JPPO
Joystick Plugs, Color Computer, Pinout

The "pinout" of the Color Computer joystick jacks is shown in Figure JPPO-1. The right and left joysticks plugs are numbered the same and have the same signals, X channel, Y channel, +5 volts, ground, and joystick switch. The jacks on the back of the Color Computer are 5-pin female DIN jacks (see BSCC). The physical pin arrangement on these 5-pin DIN jacks and plugs is different from the audio type 5-pin DIN plug available at Radio Shack (274-003) as shown in the figure. (The latter fits the cassette plug, but does not fit the joystick plug.) You can get the 5-pin male plug to fit the jack at very well-equipped electronics stores (EPWT) — the type that has been around for 30 years and carries every conceivable component.

Another option: Buy a joystick assembly (RS 26-3008) and cut and strip one or both of the cables. You may use the joystick port for reading in "analog" signals as described in ADIC or for reading in the switch inputs, as described in DICC.

Figure JPPO-1 – *Color Computer Joystick Jack Pinout*




* LOOKING IN TO FEMALE JACK ON REAR OF CC
** JOYSTK (X) READS 0 FOR 0 VOLTS UP TO 63 FOR +5 VOLTS
THIN METAL WALL PREFERRED
PIN SPACING FOR JOYSTICK CONNECTOR
(FRONT VIEW)

notes

KDFW
Killing a Disk File Without a Password, Model II/III TRSDOS, Model I/III LDOS

Use the PURGE command. Answer Y only to the file that you wish to kill. However, the catch is that you must know the disk *master password*! You are asked for it in the operation of PURGE. See FNMH for explanation of passwords.

notes

L13L
Model I/III LDOS, How to Load

1. Turn on computer (see TOCH).

2. Insert LDOS diskette (DINS).

3. Press RESET (RSWI).

4. The screen should clear and you should see a huge LDOS logo on the screen, followed by the prompt DATE?.

5. Enter the date by typing in 05/23/82 followed by ENTER, or similar date. You must have two digits for month, day, and year.

6. If you've entered the date correctly, you'll see the date digits turn into the day of the week, month, day, and year. (I'll have to admit that this was the thing that *really* sold me on LDOS, the day of the week conversion!).

7. You'll now see LDOS READY, and you're set to enter LDOS commands (see CWTC).

L3ER
L3 ERROR

Level III error (read "Disk BASIC Only" error).

Come on! You can't expect to execute Disk BASIC commands without having a disk system! Thought you could save the money, eh?

LBAP
Listing BASIC Programs, All Systems

Use the LIST command to list the BASIC lines on the display, or the LLIST (except Model I, Level I) to list on the system line printer. LIST nnn,mmm as in LIST 100-300 lists all specified lines. Use the minus sign (-) to list all lines up through or from a specified line, as in LIST -300 (beginning through 300) or LIST 300- (line 300 through end).

See TSPD for stopping the display.

LBWT
LET, In BASIC, What to Do About It, All Systems

Forget about it. LET was put into the I/II/III/CC BASIC to make the BASIC compatible with earlier BASIC versions that required the LET, as in "100 LET A=1.234". None of the Radio Shack BASICs require it, and you can simply use "100 A=1.234".

LDF1
LISTing a Disk File, Model I TRSDOS

The LIST command gives you the ability to "list" or display a disk file on the screen. Unfortunately, Model I TRSDOS will only work with ASCII files (see AFWA). If the file to be listed is not in ASCII or displayable format, the non-ASCII characters will, in technical terms, mess up the screen, and you will not be able to see a coherent display of the content of the file.

To LIST an ASCII file, do

```
LIST name
```

where "name" is a valid file name (see FNMH). An example is

```
LIST ACCTS/FEB.SECRET:Ø
```

If the file has no password, then only the name and any extension are required. The disk drive number is also optional, unless there is more than one file of the same name.

A typical LISTing is shown in Figure LDF1-1.

To stop the display, hold down the SHIFT and @ keys simultaneously. Restart by pressing any key.

Figure LDF1-1 – *Disk File LISTing, Model I TRSDOS*

```
DOS READY
LIST MAILER
BURNETT*ROGER*12311 OAKHURST*ONIONVILLE*CA*92666          }  MUST BE
CHAPMAN*PERCY*222 GRUNT CORNERS*HICKSVILLE*CA*9Ø4Ø4       }  ASCII FILE
DIOGENES*STAVOS*45.LANTERN DRIVE*PARVENUE*CA*99999        }  TO DISPLAY
FOG*PATCHY*C/O BALL PLAYERS REST HOME*WEST PUD*CA*9Ø5Ø5.  }  PROPERLY
*****

DOS READY
```

LDF2
LISTing a Disk File, Model II TRSDOS

The LIST command gives you the ability to "list" or display a disk file on the screen or printer. The file is listed in ASCII format (see AFWA) and hexadecimal format (see HNIB) at the same time. It can also be listed at a slow rate.

To list the file on the screen, do

```
LIST name
```

where "name" is the file name (see FNMH). The resulting display is shown in Figure LDF2-1. Both hexadecimal values and ASCII values are displayed. If the hex value has an equivalent *displayable* code, it is displayed, otherwise a period in the corresponding ASCII position indicates that there is no ASCII character for the code.

Displaying too fast for you? List by

```
LIST name SLOW
```

Another way to slow it down: Use the HOLD key alone to stop the display, HOLD to continue, and BREAK or ESC to exit back to TRSDOS.

To LIST to the printer, use the PRT option:

```
LIST name PRT
```

On large files, it may be tedious to list the entire file to see a record that is near the end. In this case, you can use the R option to start the LISTing from any point. The format is

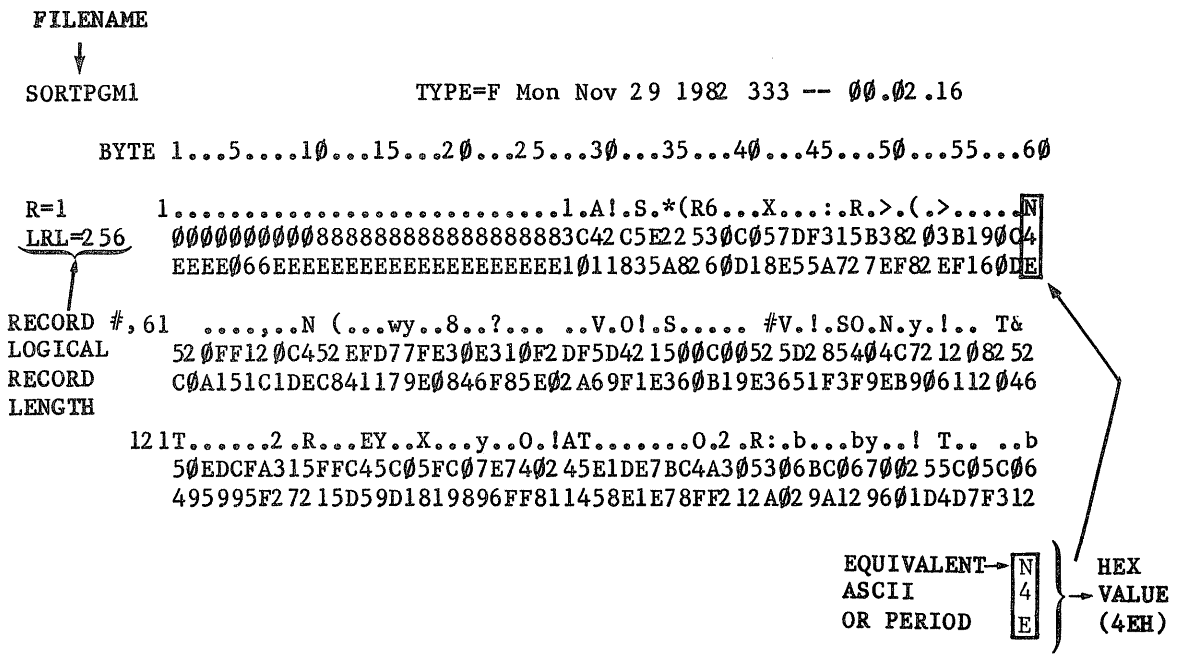
```
LIST name R=nnnnn
```

where nnnnn is a record number in the range of 1-65535. Use the DIR listing (see DLM2) to see how many records are in the file, and about how far in to start. To list on the printer from the 20th record, you'd have:

```
LIST ACCTS R=2Ø,PRT
```

You can intermix PRT, SLOW, and R in any combination.

Figure LDF2-1 – *Disk File LISTing, Model II TRSDOS*



```
SORTPGM1                    TYPE=F Mon Nov 29 1982 333 -- ØØ.Ø2.16

        BYTE 1...5....1Ø...15...2Ø...25...3Ø...35...4Ø...45...5Ø...55...6Ø

R=1        1...........................1.A!.S.*(R6...X...:.R.>.(.>.....N
LRL=256     ØØØØØØØØØØ888888888888888883C42C5E22530CØ57DF315B382Ø3B19ØC4
            EEEEØ66EEEEEEEEEEEEEEEEEEE1Ø11835A826ØD18E55A727EF82EF16ØDE

         61  ....,..N (...wy..8..?...  ..V.O!.S..... #V.!.SO.N.y.!.. T&
            52ØFF12ØC452EFD77FE3ØE31ØF2DF5D4215ØØCØØ525D285404C7212Ø8252
            CØA151C1DEC84117 9EØ846F85EØ2A69F1E36ØB19E3651F3F9EB9Ø6112Ø46

        121T......2.R...EY..X...y..O.!AT.......O.2.R:.b...by..! T.. ..b
            5ØEDCFA315FFC45CØ5FCØ7E74Ø245E1DE7BC4A3Ø53Ø6BCØ67ØØ255CØ5CØ6
            495995F27215D59D1819896FF811458E1E78FF212AØ29A1296Ø1D4D7F312
```

LDF3
LISTing a Disk File, Model III TRSDOS

The LIST command gives you the ability to "list" or display a disk file on the screen or printer. The file can be listed in ASCII format (see AFWA) or hexadecimal format (see HNIB) or both. It can also be listed at a slow rate.

To list the file on the screen in ASCII and hexadecimal format, do

```
LIST name
```

where "name" is the file name (see FNMH). The resulting display is shown in Figure LDF3-1. Both hexadecimal values and ASCII values are displayed. If the hex value has an equivalent *displayable* code, it is displayed, otherwise a period in the corresponding ASCII position indicates that there is no ASCII character for the code.

If the file is a pure ASCII file, you can list it by:

```
LIST name (ASCII)
```

Displaying too fast for you? List by

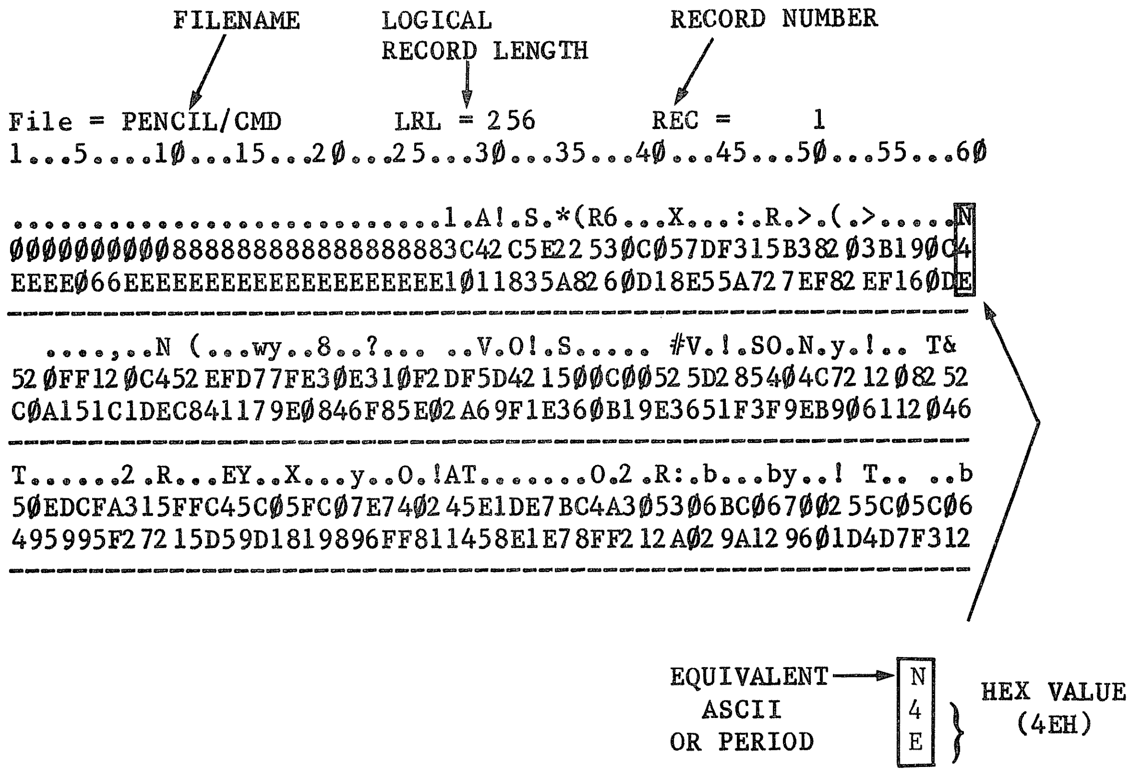
```
LIST name (SLOW)
LIST name (SLOW,ASCII)
```

Another way to slow it down — use the @ key alone to stop the display, ENTER to continue, and BREAK to exit back to TRSDOS.

To LIST to the printer, use the PRT option and an optional ASCII:

```
LIST name (ASCII,PRT)
```

Figure LDF3-1 -- *Disk File LISTing, Model III TRSDOS*



```
File = PENCIL/CMD          LRL = 256          REC =     1
1...5....1Ø...15...2Ø...25...3Ø...35...4Ø...45...5Ø...55...6Ø

...........................1.A!.S.*(R6...X...:.R.>.(.>.....N
ØØØØØØØØØØ888888888888888883C42C5E2253ØCØ57DF315B382Ø3B19ØC4
EEEEØ66EEEEEEEEEEEEEEEEEEEE1Ø11835A826ØD18E55A727EF82EF16ØDE
------------------------------------------------------------
  ....,..N (...wy..8..?... ..V.O!.S..... #V.!.SO.N.y.!.. T&
52ØFF12ØC452EFD77FE3ØE31ØF2DF5D4215ØØCØØ525D285404C7212Ø8252
CØA151C1DEC84117 9EØ846F85EØ2A69F1E36ØB19E3651F3F9EB9Ø6112Ø46
------------------------------------------------------------
T......2.R...EY..X...y..O.!AT.......O.2.R:.b...by..! T.. ..b
5ØEDCFA315FFC45CØ5FCØ7E74Ø245E1DE7BC4A3Ø53Ø6BCØ67ØØ255CØ5CØ6
495995F27215D59D1819896FF811458E1E78FF212AØ29A1296Ø1D4D7F312
------------------------------------------------------------
```

LDHT
Logic Diagram, How to Read

Want to find out what's happening in your Color Computer, Model I, II,or III? It's possible, by "reading" the logic diagram of the system and comparing it with software functions. Read DLSC for a short course in digital logic.

The logic diagram is essentially a "map" of the way the electronic circuits of the system are connected. To understand logic diagrams, then, you've first got to understand the map legend.

See Figure LDHT-1. This is a portion of the logic diagram of the Color Computer. (Some people would call it a "schematic," an older term for electronic circuits when there were more "discrete" parts such as resistors (see RHTU) or capacitors that performed more rudimentary functions, rather than containing integrations of functions). We'll use this diagram to show you how to decode the logic.

The first thing you'll see are large rectangles defining *integrated circuits* (ICs). The integrated circuit may perform a very simple function, such as providing higher current, or it may be very complex, such as a complete microprocessor chip. Generally, though, the functions within the integrated circuit are left undefined, and the interconnections of the logic diagram stop at the "pin numbers" of the integrated circuit.

Integrated circuits used in computer systems may have 8, 14, 16, 24, or 40 pins, or more. Typical sizes are 14, 16, 24, and 40 pins. The numbers you see on the outside of the lines going into the integrated circuit are the pin numbers. On the 6821 PIA chip, for example, pins 3, 4, 5, 6, 7,8, and 9 go to the MC14050B "triangles" near the left center of the page.

Not all pin numbers of the integrated circuit have to be specified in the rectangle defining the integrated circuit. Sometimes the "power pins" are not specified, as most people referring to the diagram would know that +5 volts and "ground" would be required by a particular integrated circuit. (See DLSC).

Figure LDHT-1A – *Sample Logic Diagram*

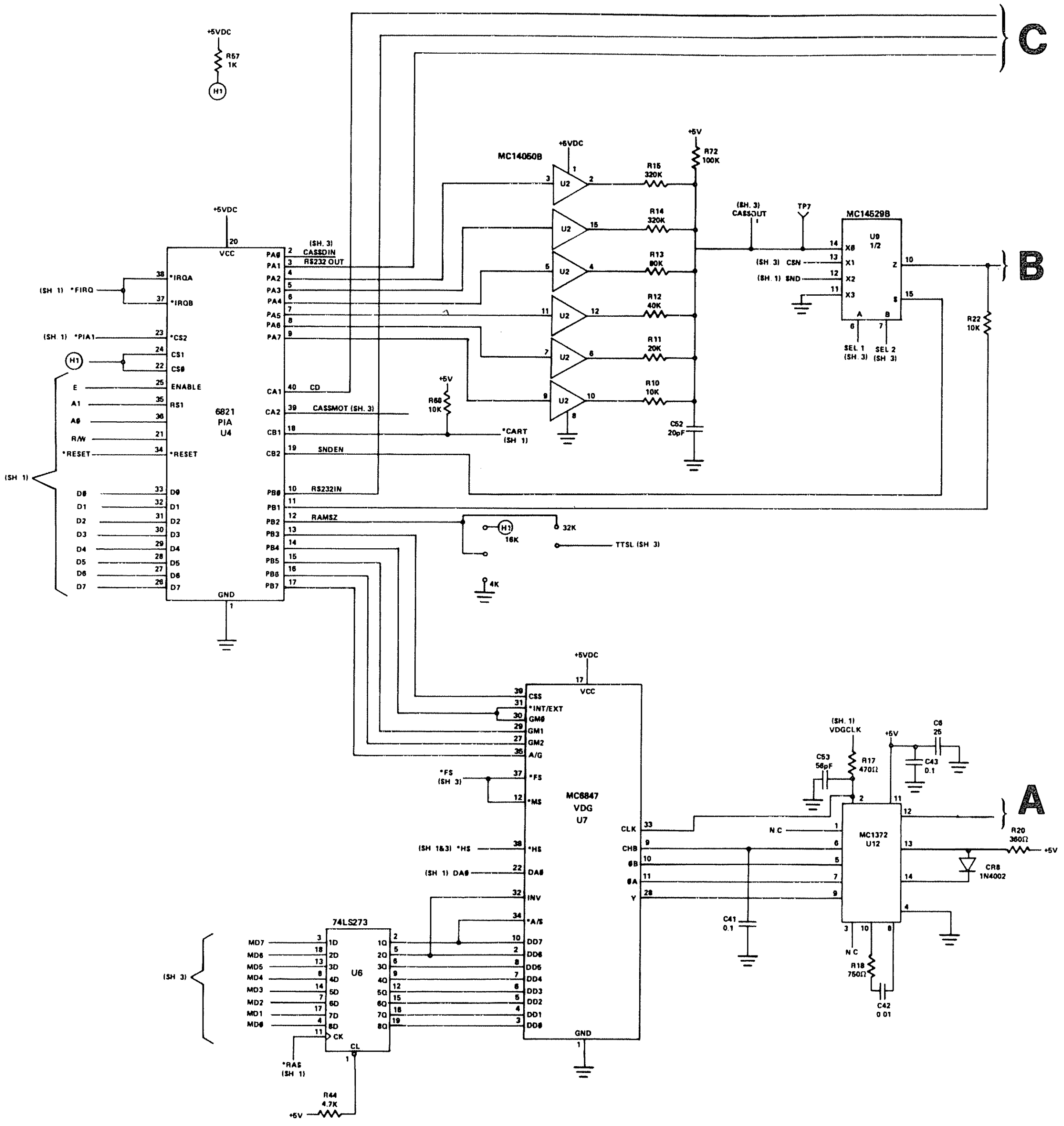
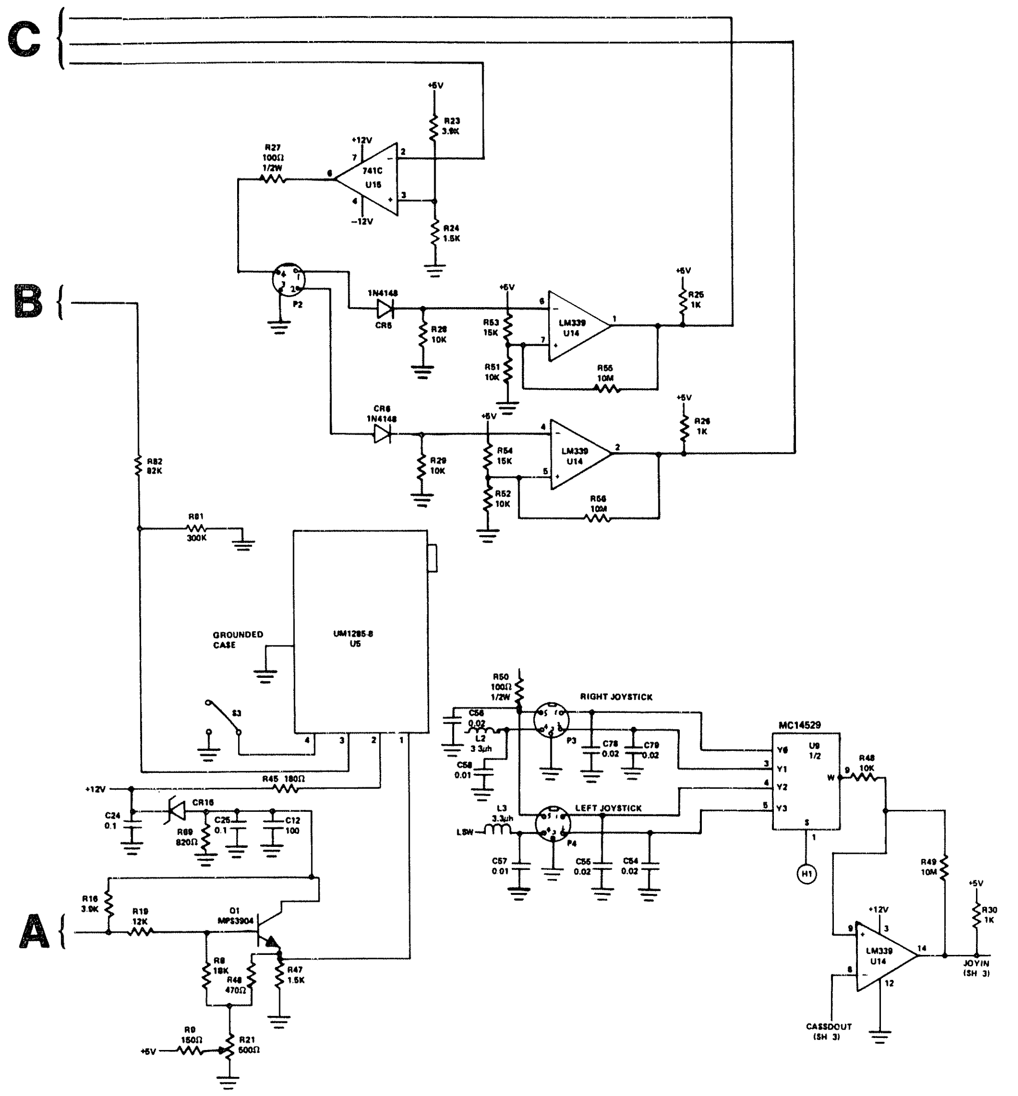


Figure LDHT-1B – *Sample Logic Diagram*



LDHT

LDHT

L

The *names* within the IC rectangle and opposite each "line" coming into the IC are the signal names for the IC pin. These signal names are defined in the IC manufacturer's specification on the IC. "ENABLE" (pin 25) of the 6821 PIA IC, for example, is defined by the manufacturer of the part, Motorola, as a signal that enables operation of the chip. Many times these "internal" signals will have different names from the "system" signal names. System signal "SNDEN," for example, goes into the CB2 input of the 6821 IC. The name on lines going out of the IC are computer system names.

Radio Shack generally uses an asterisk (*) to define that a logic signal is "active low." There are two logic levels in most digital circuits, near 0 volts and near +5. In practice, anything higher than about 3 volts is considered a +5 volt (logic 1) level, and anything lower than about 0.5 volts is considered a 0 volt (logic 0) level. Zero volts is "ground" and is represented by three parallel line segments forming a triangle.

Generally, signals are "active high" — they represent the logic condition when they are near +5 volts. When signal "D0" (extreme left) is logic 1 (+5 v), for example, there is a "1" bit on data bus line D0. However, if the signal is marked with an asterisk, the signal is active low. The *RESET signal, for example, resets the 6821 PIA when it falls to near 0 volts; normally it is near +5 volts. Another way of representing an active low signal is by a small circle on the input, as shown on the 74LS273 IC CL input in the lower left of the figure.

Logic signals shown on the logic diagram usually are named for the logic condition they represent. "SEL 1," for example, is the first "SELect" input to the MC14529B IC. A "DX" signal name is almost always "Data Bus X." D5, for example, is data bus line 5. An "AX" signal name is almost always "Address Bus X." A1, for example, is address bus line 1.

Signals generally flow from left to right and from up to down. Signals may be interconnected from page to page; in this case there is probably a "(SH. 1)" or "(PG. 5)" reference near the signal name.

How do you know when signals are "inputs" and when they are "outputs?" Generally only by knowing the functions of the ICs. (See DLSC.)

There are two broad classifications of ICs used in the Model I, II, III, and Color Computer, general purpose and special purpose.

General-purpose integrated circuits, such as the 741C and LM339, are not tailored to a specific computer, but are available to perform hundreds of applications in many types of electronic circuits. You can find descriptions of these general-purpose ICs in the Radio Shack "Semiconductor Reference Guide" (RS 276-4006), available at any Radio Shack store. While you are at it, pick up a copy of Forrest Mims book "Engineer's Notebook" (RS 276-5002). It provides lucid examples of the use of these common ICs. Radio Shack carries common ICs, but there are other general-purpose ICs that they do not carry. For descriptions of these ICs, you'll have to get general reference manuals for manufacturer's parts. You can find these reference manuals at many hobby-type electronics parts stores. Manufacturer's manuals to look for: National, Texas Instruments, Motorola.

Special-purpose IC's are specialized integrated circuits designed for microcomputer applications. The 6821 PIA, for example, is designed to work with the 6809E microprocessor chip used in the Color Computer, as is the MC6847 VDG. To get detailed specifications on these, you again must go to the manufacturer's manuals. If the IC is *really* specialized, Radio Shack may include a manual on the chip in their documentation. The Model II Technical Reference Handbook, for example, also contains detailed specifications on the Western Digital WD1791 Floppy Disk Controller IC.

For specifications on the Color Computer parts, get the Motorola "Microcomputer Data Library;" it describes all of the "68XX" parts. For specifications on the Model I, II and III, get Zilog Z-80 Microprocessor specifications. Get manuals on general-purpose ICs for all systems.

Parts not shown as rectangles are generally "discrete" components that perform simple functions.

R10 through R15 in the figure are "*resistors*" (see RHTU), and are represented by a zig-zag line marked "RX." Resistors limit current flow. The resistance value is in "ohms," and the "K" indicates multiples of 1000 (not 1024!). When resistors are connected to +5 volts (CL input of 74LS273, lower left), they are "pull ups" and generally represent a logic 1 level. When resistors are connected to ground, they are "pull downs" and generally indicate a logic 0 level. Two resistors connected in series with a line coming off the connection point are generally "voltage dividers" and produce an "analog" voltage level between +5 and 0 volts.

A "*potentiometer*" is a special case of a voltage divider, and is represented by a resistor signal with a right-angled arrow, as shown in the lower middle "pot" R21. The potentiometer is used to vary an input signal level.

Although most parts in computer systems are "digital" parts, and operate with logic 0 and 1 (0 volts and +5 volts), some parts are "*analog*" and amplify or compare signals that are intermediate values and not logic 0s or 1s. The LM339 comparators on the right of the logic diagram, for example, work with the RS-232-C inputs (see RSWI) and convert voltage level inputs between -25 and +25 volts to digital levels of +5 volts or 0 volts.

Capacitors store electrical energy. They're represented on logic diagrams by two parallel lines marked "CX," as in the case of C42 on the lower middle of the figure. The value marked next to the capacitor is normally in "microfarads," a measure of capacity. Capacitors are used to "bypass" noise to ground (C54, C55, etc) to avoid false signals, to change direct-current levels into an alternating-current signal, or to generate "clock" signals.

Diodes are devices that pass current only in one direction and are represented by a small triangle with the point going to a short line segment (see CR5 and CR6). Diodes "rectify" an alternating current to a direct current. In the case of the LM339 ICs, diodes are used to pass the positive RS-232-C input while blocking the negative RS-232-C input.

A "*zener*" *diode* (CR16, right middle) is a special case of a diode, used to establish a fixed reference voltage.

Inductances are rarely seen in computer circuits and are used primarily to block high-frequency noise, as in the case of L2 and L3 (middle right).

Transistors, such as Q1 (lower middle) are used to amplify signals, primarily for analog signals, as in the case of the "modulator" signal for the television output to the UM1285-8 part.

Larger triangles are one of several device types. The triangles shown for the MC14050B are "*buffers*". They provide more power for a logic 0 or 1; the resulting output is logically equivalent — 0 volts is still 0 volts and +5 volts is still +5 volts.

If the triangle for the buffer has a small circle on its output (not shown), it is an "*inverter*" and generates 0 volts for a +5 volt input and +5 volts for a logic 0 input.

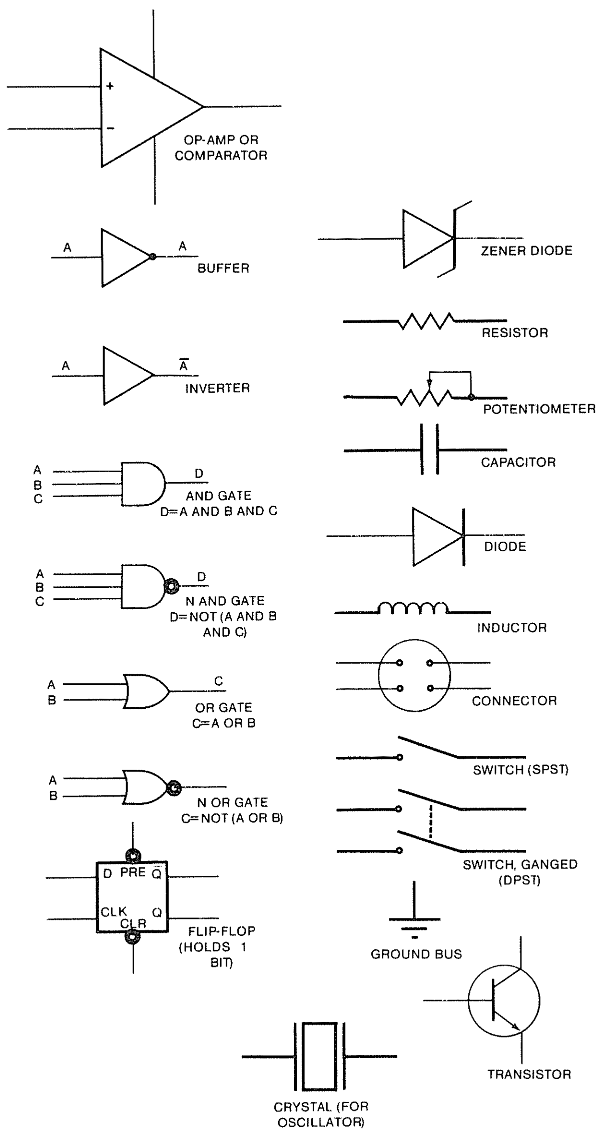
If the triangle has two inputs (left side) and one output (right side), as shown in the LM339 case, it is generally a "*comparator*". In this case, two inputs are compared. If the - input is a lower analog voltage than the + input, a logic 1 (+5 volts) is produced for the output signal. If the reverse is true, a logic 0 (0 volts) is produced for the output signal. Many times a comparator will translate a single analog input to a digital value, as is the case here in converting the RS-232-C signal of -25 through +25 volts to a digital 0 or 1.

If the triangle has two inputs, one output, and a +12 and -12 input at the top and bottom side, it is an "*operational amplifier*" that converts a digital signal to a different voltage level. The 741C (right top) converts the digital input into +12 volts or -12 volts for output to the RS-232-C device.

These and other symbols used in Model I, II, III and Color Computer logic diagrams are shown in Figure LDHT-2.

See DLSC for a description of simple digital logic devices.

Figure LDHT-2 – *Logic Diagram Symbols*



LDHT

LDHT

L

LDIS
LDOS Device Independence and System Devices

Model I/III LDOS places great emphasis on "device independence." The main thrust of device independence is this:

1. The system is organized into "logical" rather than "physical" devices.

2. Software drivers for physical devices exist as complete modules with an associated "device control block" or DCB.

As an example of a system which is not "device independent," consider a word processing program that uses the line printer. Scattered throughout the program are 11 different places at which the word processing program outputs to the line printer. Each place has several machine language instructions that load in status from the line printer, test for a "ready," and then output a string of characters.

Now suppose that you wanted to change the line printer from a standard system "Centronics" or parallel printer to a "serial" NEC Spinwriter. What would you have to do to make the change? In this hypothetical example, you'd have to go in and "patch" those 11 occurrences of instructions to address the line printer to instructions that would address a serial device. You'd have to labor long and hard to do that because you probably wouldn't have the original "source" listings to see what the instructions were. Even if you had the instructions, it would be a tedious assembly language programming job, to say the least.

In a device independent system, however, the "drivers" for each physical device, such as a line printer, would exist as code in one place. Furthermore, you could easily redefine the line printer as an "RS-232-C" device by a DOS command such as

```
ROUTE *PR TO *CL
```

which would "route" the system "printer" output to the system "communication line" device. The system would make the link and transfer all printer output to the communication line driver.

In a device independent system, you can easily route any system device to another system device, including disk files. You can also redefine the "physical" devices to different configurations, as might be the case in upgrading a 5 1/4" single-sided disk drive to a double-sided disk drive.

LDOS has the following logical device designations:

```
*KI Keyboard input (Model I/III keyboard)
*DO Display output (video)
*PR Printer
*JL Job log (log of commands with time)
*CL Communications line (RS-232-C port)
```

The device designations are used in the FILTER, ROUTE, SET, LINK, SYSTEM, and RESET commands to link one logical device to another, to substitute one device for another, to substitute a disk file for a device, or to "filter" data that is transferred between the system and the logical device.

Using these commands, you can easily do things such as:

• Route the printer output of the assembler to a disk file so that it can be processed by Scripsit.

• Dump data communication screen display to a disk file.

• Switch keyboard input so that it operates off of the RS-232-C remote input, rather than user local entry.

• Change BASIC display screen output to printer output for applications programs.

LDLD
LINKing Devices, Model I/III LDOS

Read RDLD for information on routing devices. To reiterate, routing makes one system logical device (see LDIS) take the place of another. LINK, however, links one device to another so that input/output data that would normally go only to one device now also goes to a second device.

```
LINK *PR *CL
```

This links the system print device to the comm line device so that when printing is done, data is also output to the RS-232-C interface.

System devices cannot be linked with disk files directly, but a ROUTE and LINK may be used with a "dummy" device as in

```
ROUTE *DU TO ACCOUNT/FEB
LINK *PR *DU
```

Close the disk file with a RESET *DU. Reset all linked devices after the linkage is over.

LDOS
LDOS - What Is It?

If you're currently using TRSDOS on your Model I or III, you might want to look into using LDOS. LDOS is a more advanced operating system than TRSDOS. It provides such things as:

- Logical Devices. The ability to define a "physical device" with a logical device name and to then ROUTE one logical device to another logical device or a disk file. This gives "device independence" and lets you, for example, save the output of an assembly listing as a disk file or dump data communication input from a Bulletin Board system onto a disk file.

- Extensive "Job Control Language," or JCL. Using JCL, you can run your system in a "hands off" mode, defining a sequence of operations to be performed under LDOS. You could, for example, load and execute a JCL file to run 5 BASIC programs in sequence without operator intervention, or permanently keep a sequence of involved commands as a JCL file on disk.

- A set of built-in utility programs and drivers for such things as a smart terminal program, multiple characters generated by single keystrokes, and keyboard translation.

- Enhanced commands for every common DOS function.

- Drivers for LOBO and other disk hardware that allows you to intermix 5 1/4", 8" and hard disk drives.

- Printer spooling to enable you to simultaneously print while performing other tasks (printer data is put in a disk buffer and output periodically, giving the effect of simultaneous operation).

- Model I version supports RS double-density modification.

- Incredible documentation!

Sound like I'm getting a cut of the profits? Not at all. It's simply a terrific system. On the negative side, it may be overwhelming for a beginner. Also, LDOS diskettes are not compatible with TRSDOS diskettes, although you can convert from TRSDOS to LDOS easily enough. (It's harder to go the other way around, however). Patches may be required for RS software, but are readily available.

My personal advice: Cut your teeth on TRSDOS, and then get LDOS for the Model I and III, especially for the Model I. Radio Shack now carries LDOS as an alternate operating system (Model I is 26-2213, Model III is 26-2214).

LEMC
Loading and Executing a Machine Language Program, Color Computer

The following method works for any machine language tape created for the Color Computer by the Editor/Assembler, by a CSAVEM command or by other means.

1. Position the cassette just prior to the file by the cassette recorder controls.

2. Execute a CLOADM command in BASIC if you don't know the file name. If you do know the file name, enter CLOADM "name" where "name" is the file name.

3. The cassette should start and the files will be checked for the given file name, if any. If other files precede the file, an S NAME will be displayed on the first line of the display. When the requested file is found (or the next file is found on a CLOADM without file name) an F NAME will be displayed on the first line of the display. After the file has loaded, BASIC should reply with OK. If you experience difficulty, see procedure CTLC.

4. Execute the program by entering EXEC, followed by ENTER. BASIC will transfer control to the execution address found in the cassette tape file. If this address has changed or is different from the address in the tape file, enter EXEC XXXX, followed by ENTER, where XXXX is the proper execution address.

LFBA
Line Format, BASIC, All Systems

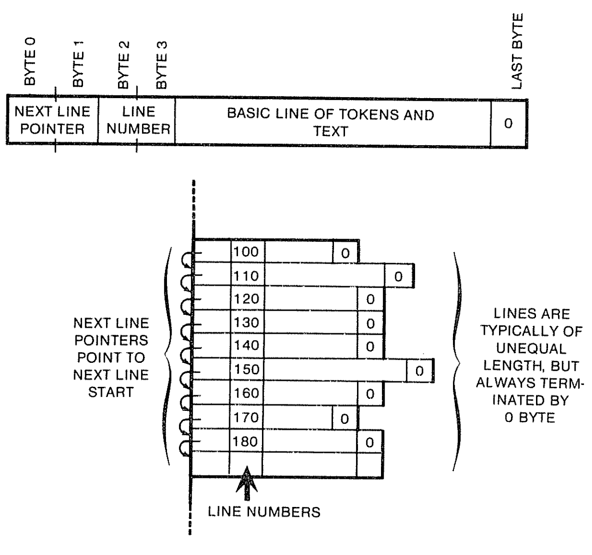
BASIC statement lines are stored in a special format, as shown in Figure LFBA-1.

The lines are arranged in numerical sequence by line number.

The first 2 bytes of each line are a "pointer" to the next line number location in memory. The 2 bytes are arranged in standard "Z-80" address format of least significant byte followed by most significant byte. The value of these 2 bytes represents the location in memory where the next line is to be found. (This location is actually directly after the end of the line, without any "gaps").

The next 2 bytes of the line are the line number in binary form, again in Z-80 address format.

Figure LFBA-1 – *BASIC Statement Line Format*


BYTE 0
BYTE 1
BYTE 2
BYTE 3
LAST BYTE
NEXT LINE POINTER
LINE NUMBER
BASIC LINE OF TOKENS AND TEXT
0
100
110
120
130
140
150
160
170
180
NEXT LINE POINTERS POINT TO NEXT LINE START
LINES ARE TYPICALLY OF UNEQUAL LENGTH, BUT ALWAYS TERMINATED BY 0 BYTE
LINE NUMBERS

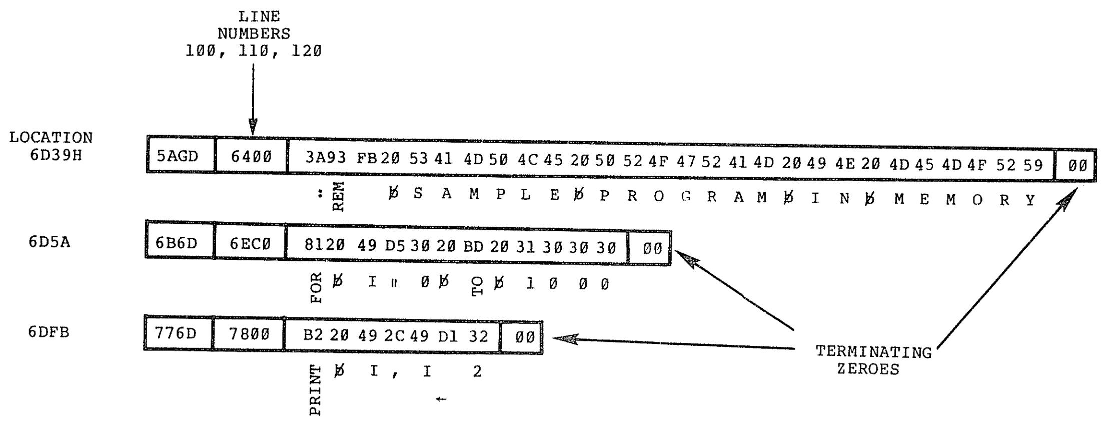
The next portion of the line is the text of the BASIC line. The BASIC statements are "compressed" into "tokens" of one byte each, so displaying the line will not result in much recognizable data.

The text portion of the line is terminated by one byte of zeroes, commonly called a "null" byte.

The last line of the BASIC program has a "next line pointer" of all zeroes.

BASIC program lines are in low memory, before the simple variables and array data (see MMM1, MMM2, and MMCC for Model I/III, Model II, and Color Computer, respectively). Figure LFBA-2 shows lines from a typical BASIC program, stored in memory.

Figure LFBA-2 – *Typical BASIC Program Lines*


LINE NUMBERS 100, 110, 120
LOCATION 6D39H
5AGD
6400
3A93 FB20 53 41 4D 50 4C 45 20 50 52 4F 47 52 41 4D 20 49 4E 20 4D 45 4D 4F 52 59
00
: REM ƀ S A M P L E ƀ P R O G R A M ƀ I N ƀ M E M O R Y
6D5A
6B6D
6EC0
8120 49 D5 30 20 BD 20 31 30 30 30
00
FOR ƀ I = 0 ƀ TO ƀ 1 0 0 0
6DFB
776D
7800
B2 20 49 2C 49 D1 32
00
PRINT ƀ I , I ↑ 2
TERMINATING ZEROES

LMFI
LOADing Multiple Machine Language Files from Disk, All Systems

No problem. Simply execute one LOAD (I/II/III) or LOADM (CC) (see LMLD) after another. The files will load as specified in the load addresses in the file. As long as you DUMPed the files with non-overlapping addresses (see DMT2, DMT3, DMTL), the files should not conflict with one another. (SAVEM is used on the Color Computer). A good example of this is a number of assembly language object modules that have been assembled at different ORGs. They can be LOADed into RAM one at a time to make up a complete program.

LMFN
Loading Machine Language Files from Cassette, Models I and III

Machine language (see MLWI) files are applications programs such as SCRIPSIT that are not written in BASIC or such things as memory DUMPs that you have created by using DEBUG. We'll describe here how to load these files on a non-disk Model I or III.

If you have a Model I, skip this paragraph. Ah hah! I'll bet you have a Model III. First look at the tape to see whether it is "500-baud" or "1500-baud." If it is a Model III tape and is not marked it will be 1500-baud. You are in the 1500-baud mode if you responded with ENTER (or "H") to the Cass? prompt whem you powered up or reset your system. You are in the 500-baud mode if you responded with "L" to the Cass? prompt. If you are not in the proper cassette speed, either power down, wait 5 seconds and power up and set the speed to match the cassette tape, or see FHBF.

Model I/III: These files are loaded in the SYSTEM mode. To get to the SYSTEM mode, simply enter

```
SYSTEM
```

after the BASIC prompt of >.

You'll then see an *? "prompt," indicating that you have arrived in SYSTEM mode.

Now load the application cassette into your cassette recorder. Connect the plugs as shown in CRPI. Rewind the cassette to the beginning, and enter the name of the cassette.

The cassette recorder should now start, and you'll see two asterisks in the upper right-hand corner of the screen. The SYSTEM cassette loader looks for the first character of the file name. If there is more than one file on the cassette, you will see the first character of each cassette file name displayed in the asterisk area. When the file you've designated is reached, the first character will disappear, and the file will load.

Loading will take from a few seconds to 5 minutes. Loading is indicated by the flashing cursor. If a "C*" is present, a checksum error has occurred. See CTLC for corrective measures.

After a valid load, the *? prompt will appear. The program has now been loaded and the next step is to transfer control to the program start. For most Radio Shack or other programs, this simply involves entering

/ followed by ENTER

The program should now start. If the cassette is marked with an indication that the starting address is at a specific location, however, you might have to enter a starting address by

/MMMMM followed by ENTER

where MMMMM is a decimal starting address. This is rarely the case.

LMLB
Loading a Machine Language File From Disk BASIC, Model III

TRSDOS or LDOS:

A machine language file can be loaded either in command mode or in a BASIC program statement by the CMD"L" command. The format is:

```
CMD"L","name"
```

where "name" is the name of the machine language disk file to be loaded or a string variable.

Normally, this would be a machine language disk file associated with a BASIC program. It cannot overlay the high-memory area protected by the MEMORY SIZE? response (see PROT).

If the filename format is used, the filename must be enclosed in quotes, as in

```
1ØØØ CMD"L","LPDRVR"
```

If a string variable is used, no quotes are required, as in

```
9ØØ INPUT "AL PROGRAM",AL$     'input program required
91Ø CMD"L",AL$                 'load it
```

LMLD
Loading a Machine Language Program or Data File from Disk, All Systems

To load a machine language file created by DUMP (I/II/III) or SAVEM (CC) or other means, use the LOAD command:

```
LOAD name     (I/II/III)
```

or

```
LOADM name    (Color Computer)
```

where "name" is a standard disk file name (see FNMH).

TRSDOS or LDOS will load the file by referring to the load addresses in the file. The operating system is not aware of what type of file it is loading, and won't make any checks for valid instructions or other criteria.

After the file is loaded, the program or data loaded will remain in the memory area specified by the load, but can be destroyed by use of BASIC or other programs.

LDOS users only: Use the (X) parameter to load a machine language file from a non-system diskette

```
LOAD (X) FLAKEY/CMD
```

In this case, the file must reside above **5300H (5200H** for standard load).

LNAT
Line Numbering, Automatic, BASIC, Models I/II/III

Not applicable on Model I/III Level I: Use the AUTO command in the command mode. If you enter AUTO alone, BASIC will start from line number 10 and increment by 10 for each new line. If you enter AUTO line #, BASIC will start from the given line number with increments of 10. If you enter AUTO line #,increment, BASIC will start from a given line number with a given increment. Example: To number starting with line 333 in increments of 17, do:

```
AUTO 333,17
```

Boy, are you *weird* . . . increments of 17 . . .

LOHT
Logical Operations in BASIC, All Systems

There are a number of **logical operators and operations** in both BASIC and assembly language.

A logical AND is represented in BASIC by AND (Model I/III Level I by "*"). An AND of two values is done on a binary level. An AND of two bits goes like this:

```
Ø AND Ø=Ø
Ø AND 1=Ø
1 AND Ø=Ø
1 AND 1=1
```

An AND produces a 1 bit only if there is a 1 bit in the first bit AND the second bit.

When ANDs are done with 8-or 16-bit arguments, each bit is ANDed separately to produce the result:

```
    Ø1Ø11111
AND ØØ111Ø1Ø
    --------
    ØØØ11Ø1Ø
```

Naturally, an AND in BASIC will produce the result in the variable, as in

```
1ØØ A=B AND C  'A=26 if B=95 and C=58, as in above
```

An OR is similar to an AND except that a 1 bit is produced in the result if either one bit OR the other is a 1. (Model I/III represents an OR by "+").

```
Ø OR Ø=Ø
1 OR 1=1
Ø OR 1=1
1 OR 1=1
```

BASIC operations are similar to the AND except that each bit position is ORed.

A NOT is used either as a relational operator or in bit manipulation. (Not applicable to Model I/III Level I).

```
1ØØ IF NOT(A=B) THEN GOTO 1ØØØ
```

In this case the GOTO will be executed if A<>B.

If variable B is 10 (binary 0000000000001010), then

```
1ØØ A=NOT(B)
```

would result in variable A being set to the *complement* of B; in this case, all 1s are changed to 0s and all 0s are changed to 1s. Variable A would be 1111111111110101, or -11 in integer format.

The Model II also uses **XOR**, or Exclusive OR. This operation is used similarly to AND and OR. The "truth table" for an XOR is

```
Ø XOR Ø=Ø
Ø XOR 1=1
1 XOR Ø=1
1 XOR 1=Ø
```

The Model II also uses **EQV**. This operation is used similarly to AND or OR. The "truth table" for an EQV is

```
Ø EQV Ø=1
Ø EQV 1=Ø
1 EQV Ø=Ø
1 EQV 1=1
```

The result bit is set when both bits are the same.

The Model II also uses **IMP**. This operation is used similarly to AND and OR. The "truth table" for an IMP is

```
Ø IMP Ø=1
Ø IMP 1=1
1 IMP Ø=Ø
1 IMP 1=1
```

BASIC operations are similar to the other logical operations; each bit position is considered individually with its own EQV or IMP operation.

Although you may find use for EQV, IMP, and XOR, the most common logical operations you'll be doing on Model II BASIC will be ANDing and ORing.

All Systems: If AND, OR, and the other logical operators are used in a *relational expression* (such as IF (A = 5) OR (B > 6) THEN . . .) they test a logical condition; if they are used with numeric operands, they operate on a bit level.

L

LP3S
Line Printer III, Operation Notes

The Line Printer III has been superseded by the Line Printer V which has been superseded by the DMP-500. There's not a great deal you can do with the Line Printer III, although it will print up to 132 characters per line and is fast. The character set contains standard ASCII characters (see ADFW) from blank (**20H**, 32 decimal) through 127 (tilde). Under program control you may:

- Change the character size from small (10 characters per inch) to large (5 characters per inch). To change from small to large characters, output a CHR$(31). To change from large to small characters, output a CHR$(30). This may be done at any time.

- Change the vertical line spacing from 6 lines per inch to 8 lines per inch. To change to 6 lines per inch, output CHR$(27)+CHR$(54). To change to 8 lines per inch, output CHR$(28)+CHR$(56). This may be done at any time.

The "power on" defaults are small characters and 6 lines per inch.

Hint on ribbons: The LP III "eats" ribbons. The replacement ribbons for the LP III cost $13.95 and last for about 100,000 characters before printing gets fairly light. One way to save on ribbon costs is to get "ribbon inserts", about $6 each. The cartridge for the LP III can be split apart and a new ribbon inserted in about 3 minutes. You may not be able to shake hands afterwards, but you've saved yourself some money. Try BCCOMPCO, Box 246, Summersville, MO 65571, or others.

LPHT
Logic Probe, How to Use

You can use an inexpensive logic probe to test the digital outputs of internal and external circuits in your system. It's easy to use and even electronics novices should have no problems. The probe will indicate a logic 0 or logic 1 (see DLSC), but, more importantly, will indicate a pulse output. I'll describe a Radio Shack logic probe (RS 22-301) here, but others are similar.

The Radio Shack logic probe is shown in Figure LPHT-1. There are two clip leads connected to the cable that comes from the end of the probe. The red lead attaches to the positive output of a power supply, while the black lead attaches to the ground output of a power supply. You may use power from your system or a separate power supply than the one associated with your computer system. If you use a separate power supply, connect the ground lead of the logic probe to the system power supply ground. See Figure LPHT-2.

Figure LPHT-1 – *Typical Logic Probe*

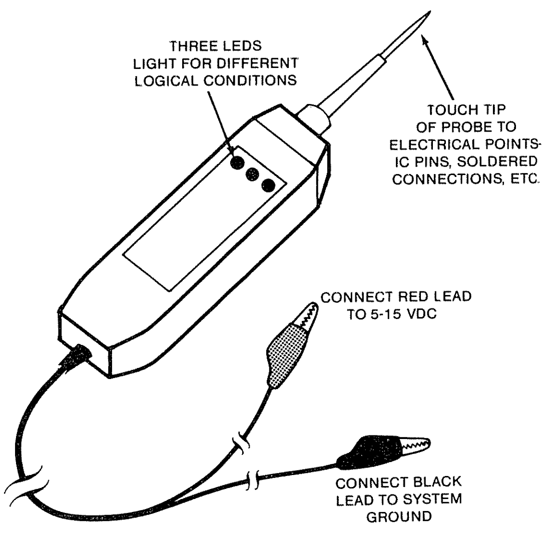
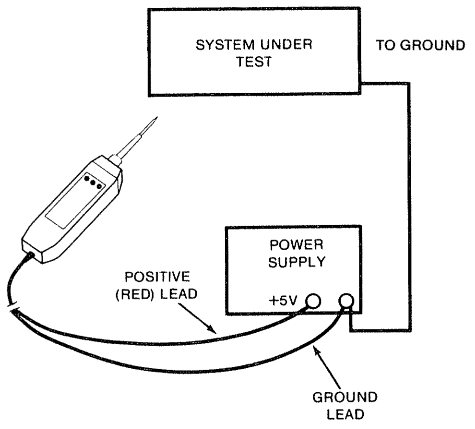


Figure LPHT-2 – *Power Supply Connections*



Once you've connected the logic probe to power, you can now touch the probe end to any IC pin or connector pin and get a reading on the logical state.

1. To test power applied to ICs:

   A. Touch probe to VCC pin(s). You should see

   ```
   LO=off PULSE=off HI=on
   ```

   B. Touch probe to ground pin(s). You should see

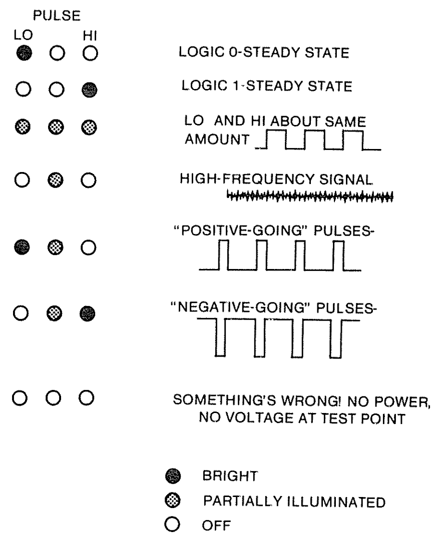
   ```
   Lo=on PULSE=off HI=off
   ```

   C. You should not see a PULSE=on condition at any time. If you do, power is not connected properly.

2. To test logic outputs: (see Figure LPHT-3)

   A. Touch probe to logic output.

Figure LPHT-3 – *Logic Probe Indications*



B. The LO or HIGH LED should light to indicate a logic 0 or 1. If you expect a constant logic level, you should not see a pulse indication.

C. If the output is alternating between 0 and 1, several cases might be true:

a. If the output is a square wave with 50% duty cycle (half the time a 1 and half the time a 0), you'll see the HI and LO leds light with the same intensity and the PULSE led blinking. This indicates a square wave of less than 100 kilohertz.

b. If the HI and LO led are off and the PULSE led is blinking, a square wave of greater than 100 kilohertz is being received.

c. If the HI led is on and the PULSE led is blinking, the signal is normally high with "negative going" pulses, as shown in Figure LPHT-3.

d. If the LO led is on and the PULSE led is blinking, the signal is normally low with "positive going" pulses, as shown in the figure.

e. If the PULSE led is on and if the LO led or HI led are both on, but are unequal in brightness, there is pulse activity with the LO or HIGH led indicating the proportion of logic 0 or 1 states. In other words, if the LO led is brighter, the signal is more often low (0), and if the HIGH led is brighter, the signal is more often high (1).

Pulses as short as 300 nanoseconds can be detected with the probe. Normally, you will not need this resolution on your system. Frequencies as high as 1.5 megahertz (1.5 million cycles per second) can be detected. Most frequencies in the four systems are lower than this, with the exception of system clocks.

LSER
Long String error.

Strings can only be 255 characters long. This is usually not a problem, unless you're trying to do something like making up dummy strings for graphics (see GHS1), or for insertion of machine language code. Consider using arrays for contiguous memory space in the latter two cases.

LWD1
Line Printer, Writing Your Own Driver, Model I

If you want to write your own line printer or character printer driver, you must use an assembly language subroutine. BASIC code will work, but will not be able to keep up with the line printer speed.

The basic "loop" for outputting a single character to any system line printer on the Model I looks like this in BASIC:

```
1ØØ ′ LINE PRINTER OUTPUT SUBROUTINE
11Ø A=PEEK(14312)              ′37E8H
12Ø A=A AND 24Ø                ′ØFØH
13Ø IF A<>48 THEN GOTO 11Ø     ′if not 3ØH
14Ø POKE 14312,B               ′output character in B
15Ø RETURN                     ′return
```

This loop checks line printer status by reading the line printer address from input/output address 14312 (**37E8H**), checking the upper 4 bits for "busy", "outpaper", "unit select" and "fault"; looping if busy or out of paper; and outputting the character in variable B if not busy or out of paper.

The assembly language code emulates this BASIC code exactly:

```
OUTLP CALL  ØA7FH         ;get character into HL
LOOP  LD    A,(37E8H)     ;get status
      AND   ØFØH          ;get upper 4 bits
      CP    3ØH           ;test for busy, out paper
      JR    NZ,LOOP       ;go if busy, out paper
      LD    A,L           ;get character
      LD    (37E8H),A     ;output
      RET                 ;go for next character
```

The relocatable machine language code for this subroutine is

```
2Ø5,127,1Ø,58,232,55,23Ø,24Ø,254,48,32,247,125,
5Ø,232,55,2Ø1
```

See CML1 or CMD1 on instructions on how to execute it from BASIC.

This code will only write one character. What about writing a string of characters? The normal method of doing this in assembly language is to write characters until a special character, usually a zero (null), is detected. A zero is added to the end of every string to be written and causes a return from the subroutine. The assembly language code looks like this:

```
OUTST CALL  ØA7FH        ;get string loc'n to HL
LOOP1 LD    A,(HL)         ;get next character
      OR    A              ;test for zero
      RET   Z              ;return if null
LOOP2 LD    A,(37E8H)          ;get status
      AND   ØFØH               ;get upper 4 bits
      CP    3ØH                ;test for busy, out paper
      JR    NZ,LOOP2           ;go if busy, out paper
      LD    A,(HL)          ;get character
      LD    (37E8H),A       ;output
      INC   HL              ;bump pointer
      JR    LOOP1           ;go for next character
```

The relocatable machine code for this subroutine is

```
2Ø5,127,1Ø,126,183,2ØØ,58,232,55,23Ø,24Ø,254,48,
32,247,126,5Ø,232,55,35,24,237
```

It will output a string of ASCII characters to the line printer until a 0 (null) is read. Refer to CML1 or CMD1 for instructions on how to execute it from BASIC.

If you want to do things such as translating from ASCII to Baudot, converting to NEC Spinwriter "escape sequences", and the like, you can use the basic subroutines above, but will have to add additional code. This is not an easy process for a beginner, but can be done. See various procedures in this book for info on using assembly language.

LWD3
Line Printer, Writing Your Own Driver, Model III

If you want to write your own line printer or character printer driver, you must use an assembly language subroutine. BASIC code will work, but will not be able to keep up with the line printer speed.

The basic "loop" for outputting a single character to any system line printer on the III looks like this in BASIC:

```
1ØØ ' LINE PRINTER OUTPUT SUBROUTINE
11Ø A=INP(248)                'ØF8H
12Ø A=A AND 24Ø               'ØFØH
13Ø IF A<>48 THEN GOTO 11Ø    'if not 3ØH
14Ø OUT248,B                  'output character in B
15Ø RETURN                    'return
```

This loop checks line printer status by reading the line printer address from input/output port 248 (**0F8H**), checking the upper 4 bits for "busy", "outpaper", "unit select" and "fault"; looping if busy or out of paper, and outputting the character in variable B if not busy or out of paper.

The assembly language code emulates this BASIC code exactly:

```
OUTLP CALL  ØA7FH        ;get character into HL
LOOP  IN    A,(ØF8H)         ;get status
      AND   ØFØH             ;get upper 4 bits
      CP    3ØH              ;test for busy, out paper
      JR    NZ,LOOP          ;go if busy, out paper
      LD    A,L          ;get character
      OUT   (ØF8H),A     ;output
      RET                ;go for next character
```

The relocatable machine language code for this subroutine is

```
2Ø5,127,1Ø,219,248,23Ø,24Ø,254,48,32,248,125,211,
248,2Ø1
```

See CML1 or CMD1 on instructions on how to execute it from BASIC.

This code will only write one character. What about writing a string of characters? The normal method of doing this in assembly language is to write characters until a special character, usually a zero (null), is detected. A zero is added to the end of every string to be written and causes a return from the subroutine. The assembly language code looks like this:

```
OUTST CALL  ØA7FH        ;get string loc'n to HL
LOOP1 LD    A,(HL)          ;get next character
      OR    A               ;test for zero
      RET   Z               ;return if null
LOOP2 IN    A,(ØF8H)          ;get status
      AND   ØFØH              ;get upper 4 bits
      CP    3ØH               ;test for busy, out paper
      JR    NZ,LOOP2          ;go if busy, out paper
      LD    A,(HL)         ;get character
      OUT   (ØF8H),A       ;output
      INC   HL             ;bump pointer
      JR    LOOP1          ;go for next character
```

The relocatable machine code for this subroutine is

```
2Ø5,127,1Ø,126,183,2ØØ,219,248,23Ø,24Ø,254,48,
32,248,126,211,248,35,24,239
```

It will output a string of ASCII characters to the line printer until a 0 (null) is read. Refer to CML1 or CMD1 for instructions on how to execute it from BASIC.

If you want to do things such as translating from ASCII to Baudot, converting to NEC Spinwriter "escape sequences" and the like, you can use the basic subroutines above, but you will have to add additional code. This is not an easy process for a beginner, but can be done. See various procedures in this book for info on using assembly language.

notes

M13H
Model I/III Hardware Differences

The Model I and III systems are more alike than different; however, programs that use assembly language and BASIC programs that use embedded assembly language routines designed for one machine probably won't run on the other machine. Here are the differences between the two systems:

1. The Model I (standard configuration) uses "single-density" disk drives. The Model III uses "double-density" drives.

2. The Model I uses a 1771 floppy disk controller; the Model III uses a 1793 floppy disk controller. Special-purpose programs that communicate directly to the disk controller for the Model I will not work on the Model III and vice versa. This would include programs such as disk diagnostics and "copy" programs that allow you to copy "protected" diskettes.

3. Software disk I/O calls use the same locations. However, the DCB on the Model I is 32 bytes, while the DCB on the Model III is 64 bytes. All BASIC disk file operations will be the same on both machines. Assembly language programs that use the "Disk I/O" calls may have to be modified to increase the size of the DCB.

4. The Model III uses an additional 2K bytes (2048) of ROM for the BASIC interpreter, as shown in Figure MMM1-1. This should normally be no problem, unless special user-designed hardware used these locations on the Model I.

5. The line printer address in the Model I is **37E8H**. Input/output is done by writing or reading to this "memory-mapped" address in the Model I. In the Model III, "I/O-mapped" I/O is done, with the I/O address of the line printer being **0F8H**. Programs that address the line printer directly without going through "ROM calls" may have to be patched to convert one type of I/O to another. This would involve "patching" of several bytes of the program.

6. RS-232-C communications for the two systems is essentially the same. The Model III adds some additional signals, but retains all previous signals.

7. The keyboard configuration is the same on the Model I and III with one exception. The Model I looks for either the left or right SHIFT key on bit 0 of address **3880H**. The Model III looks for the right SHIFT key on bit 1 of address **3880H** and the left shift key on bit 0 of address **3880H**. Programs that scan the keyboard without going through the ROM calls may have to be modified for this change, or one SHIFT may work while the other may not.

8. External I/O devices that attach to the system I/O bus have to be "enabled" by setting bit 4 at I/O address **0ECH**. The external I/O device response is also different. Writes can be done without this response enable, but reads will not work.

9. Cassette tape I/O will work properly as long as the ROM calls for cassette I/O are used. If reading 500-baud Model I BASIC tapes on the Model III, set the 500-baud mode by doing a POKE 16913,0. Programs that interface directly to the cassette I/O port at I/O address **0FFH** will have to be changed (not many programs will do this).

10. The Model III provides more system interrupts than the Model I.

11. The Model III display uses a video "wait" logic so that the video display memory can be updated only during periods of no screen display (retrace). This prevents flicker, but should have no impact upon software.

12. The Model I 32/64 character mode is defined by bit 3 at I/O address **0FFH**; the Model III 32/64 character mode is set by bit 2 at I/O address **0ECH**. This may affect assembly-language programs that set the bit directly.

13. The Model I cassette recorder is turned on or off by setting and resetting bit 2 of I/O address **0FFH**; the Model III 32/64 cassette recorder is turned on and off by setting and resetting bit 1 of I/O address **0ECH**. This may affect assembly language programs that change this bit directly.

14. The Model I may use two recorders. The Model III allows only one recorder.

15. The "published" and supported ROM I/O calls remain the same (see RCWA). ROM has been changed, however, and "non-standard" (i.e., unpublished) ROM calls used in Model I software may not work on the Model III and vice versa.

M1ON
Modem I, Operation Notes

The Modem I is a Radio Shack modem, used to link a TRS-80 Model I, II, III, or Color Computer to another computer system via the telephone lines. See "Modems, How to Use" (MHTU) and "Modems, What Are They" (MWAT).

To Use the Modem I with Model I, Model II, Model III with RS-232-C

1. Connect the modem to the system via a standard 25-conductor RS-232-C cable. You'll need a male "DB-25" connection on the Modem I end, and a male on the other end, except for the Model I case, where you'll need the original cable that came with the RS-232-C board, an "edge connector" to male DB-25 (RS 26-1145).

If you do not have the cable, you cannot go wrong in using a 25-conductor cable, either the Radio Shack version (RS 26-4403 for Model II or 26-1408 for Model III).

2. Connect a telephone into the standard telephone jack in back of the modem. Radio Shack sells conversion plugs to convert other phone plugs into this type of connector.

3. Set the RS-232/CASS switch on the back of the Modem I to RS-232.

4. Set the ANS/OFF/ORIG switch on the front of the Modem I to OFF. Plug the telephone plug from the Model I into the phone jack in your wall (all right, so it's on the floor!).

5. Pick up the telephone. You should hear a dial tone even with the Modem I power off.

6. Plug the power adapter into 110 VAC, and connect the output power plug to the pin jack on the modem. Momentarily switch the ANS/OFF/ORIG switch to ORIG; the ON indicator should light; if it does not, you are not getting power. The ON light should be on in either the ANS or ORIG mode. Set the ANS/OFF/ORIG switch back to OFF.

7. You're now ready to use the modem. Load and start your data communications software.

8. Call up a bulletin board (BBUS), CompuServe (CPSU), or other communications system. Here are some further notes:

A. Dial the number by using the telephone with the Modem I switch at OFF.

B. When you hear the "carrier" at the other end of the line, switch the ANS/OFF/ORIG to ORIG. In certain situations, you will want to be in the ANS mode, but the normal setting will be ORIG. See MWAT.

C. Hang up the phone.

D. The CD (carrier detect) light on the face of the modem should be on. It indicates that you are receiving the tone from the "other end" of the line. When this light goes off, the other system has disconnected.

E. You should now be communicating; follow the procedures in BBUS or CPSU.

To use the Modem I with the Color Computer:

1. Connect the Modem I with Radio Shack cable 26-3020. This is a 4-pin DIN plug to 4-pin DIN plug cable, wired "straight-through." Make up your own if you'd like (see RCCC). Plug the Color Computer end into the RS-232-C plug (see BSCC) and the Modem I end into the 4-pin DIN connector. There is no "wrong end".

2. Follow steps 2 through 8 above to start communications.

To use the Modem I with Model I Level II computer without an RS-232-C (you're kind of a cheap son-of-a-gun, aren't you?), you'll need a special software package (RS 26-1139):

1. Connect to the Modem I with a special 5-pin DIN to 4-pin DIN cable (RS 26-3009). The 5-pin end plugs into the cassette output plug on the rear of the CPU (see BSCC) and the 4-pin end plugs into the 4-pin jack of the Modem I.

2. Set the RS-232/CASS switch on the rear of the Modem I to CASS. This switch enables conversion of the low-level cassette signal to a form palatable to the Modem I.

3. Do steps 4 through 8 above to start communications.

The Modem I is basically a 300-baud, direct connect modem that operates in full-duplex (except for the Model I, cassette version, which is not true full-duplex). For explanations of these terms, see RSWI.

There is no "programming" of the modem as is discussed in MWAT — it just sits there and converts data, humming away in both directions.

MFAL
Memory, Finding Amount Left

In BASIC command mode, do a PRINT MEM to find the number of bytes left in the user RAM. MEM can also be used inside a BASIC program to dynamically check on the memory remaining.

MHTU
Modem, How to Use

1. Read procedure MWAT if you know nothing about modems.

2. Do you have an acoustic coupler (with two cups for the telephone headset)? If so, go to procedure ACHT. If not, continue.

3. Connect a cable between the 25-pin RS-232-C connector on the modem and the 25-pin RS-232-C connector of the Model I, II (B), or Model III. This cable is available from Radio Shack (26-1408) or other electronics/computer stores. Connect a special cable (RS 26-3014) between the Color Computer RS-232-C port (4-pin DIN, see BSCC) and the 25-pin RS-232-C connector of the modem; some Radio Shack modems will have special 4-pin DIN plugs, and you can use the 4-wire RS cable 26-3020.

4. Model I: Setup your COMM/TERM and Sense Switches as described in procedure RHS1. Model II: Setup the RS-232-C interface as described in the RS manual. Model III: Setup the RS-232-C interface as described in procedure RHS3. Color Computer: Setup the RS-232-C interface as described in the RS manual.

5. You're now ready to dial a Bulletin Board system, CompuServe, or other data communications system. Setup your modem this way:

A. If you are originating the call, set the modem switch for Originate/Answer to "Originate." Sometimes this switch will be labeled "O/A."

B. Set the modem switch for "Full/Half" to "Full" if your program uses full-duplex (see RSWI) or "Half" if your program uses half-duplex. Typical use is full-duplex to "echo-back" characters transmitted.

6. Load and start your data communications program. Simple programs will now be waiting with a blank screen, ready to receive data and display it, and to transmit data from the keyboard.

7. Dial up the number of the Bulletin Board or network while listening on the audio output of the modem. If your modem does not have an audio output, you'll need a parallel connection of a normal telephone to hear the tone. Radio Shack sells a phone plug with two outlets to enable you to connect the modem and telephone to the same outlet ("duplex jack" RS 279-357). Some modems have a jack for a telephone built in (see M1ON for example). Another thing you *can* do: Dial the number without being able to hear the audio response, and take your chances, but this is not a good, permanent mode of operation. After the phone is answered, you hear a pause followed by a high-pitched whine of the "carrier" frequency. Hang up the phone if you're using one.

8. You should now see the prompt message of the Bulletin Board or network on the screen. If you do not, try typing ENTER a few times. If you still see nothing, check the "carrier" light on the modem. If it is off, the other system has "disconnected" you. (It will "time out" if it does not get the proper response in time.) Review the steps above, and try again with another system (preferably). If you still have trouble, refer to procedure RSH1 or RSH3. If you do see data on the screen, but it's garbled, go to procedure RSH1 or RSH3 to check the word length, stop bits, and baud rate settings. If you see meaningful data, continue with the procedure for Bulletin Boards (BBUS) or CompuServe (CPSU).

MLWI
Machine Language, What Is It?

Every microcomputer contains a microprocessor chip that is the heart of the microcomputer. The microprocessor chip is the "central processing unit" of the microcomputer and contains most of the electronic logic for the "processing" portion of the computer system. The microprocessor chip in the Models I, II, and III is the Z-80A; the microprocessor chip in the Color Computer is the 6809E. The two microprocessors are roughly comparable in speed and power.

The microprocessor chip has a built-in "instruction set," which is simply a set of instructions that it will perform. There are hundreds of separate instructions for the Z-80A and 6809E, but they are very rudimentary — such things as "Add Two 8-Bit Numbers" and "Branch if Two Values are Equal." All programs are ultimately made up of a "machine language" program that strings together thousands of these "machine" instructions and executes them at hundreds of thousands of instructions per second. You may think you're programming in BASIC, but the Z-80A or 6809E is actually executing the machine language instructions of the BASIC interpreter program in ROM (read-only-memory).

You may also talk directly to the Z-80A or 6809E by creating your own machine language programs. You can do this by studying machine language by a book such as Radio Shack's "*TRS-80 Assembly Language Programming*" by William Barden, Jr. (no relation), or by using Radio Shack or other "assembler programs," which will convert a special language called "assembly language" into the proper machine language instructions.

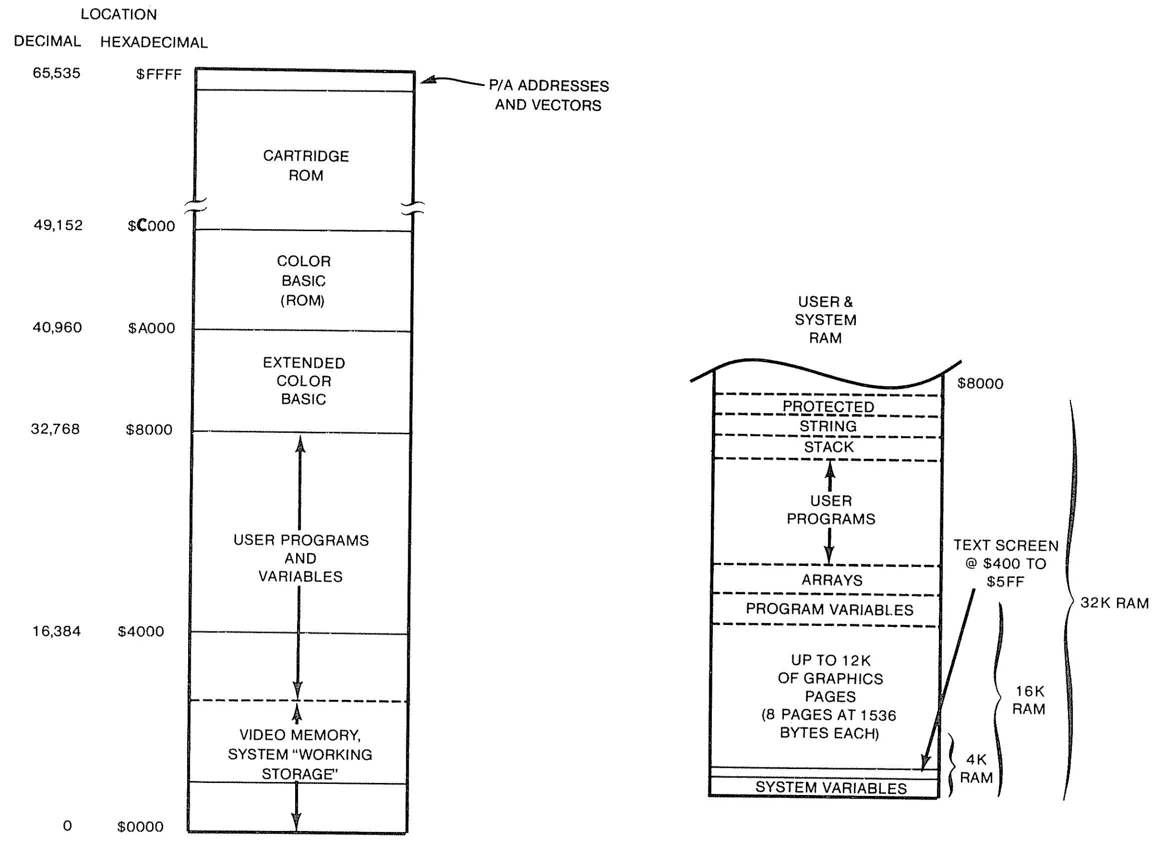
For information on using machine language, see CML1, CMD1, and CMLC. For information on using assembly language, see ALWI.

MMCC
Memory Map, Color Computer

Figure MMCC-1 shows a memory map of the Color Computer.

(See next page).

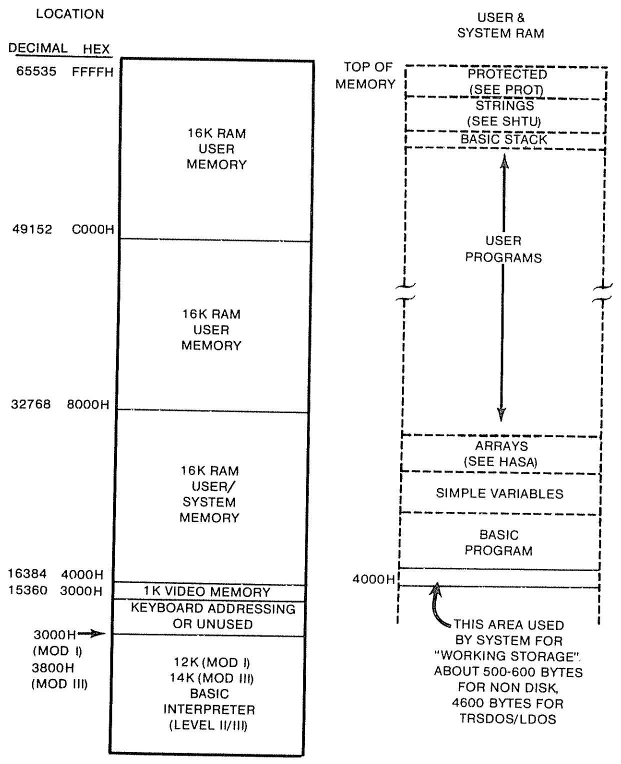
Figure MMCC-1 – *Memory Map, Color Computer*



MMM1
Memory Map, Model I/III

Figure MMM1-1 shows a memory map of the Model I/III.

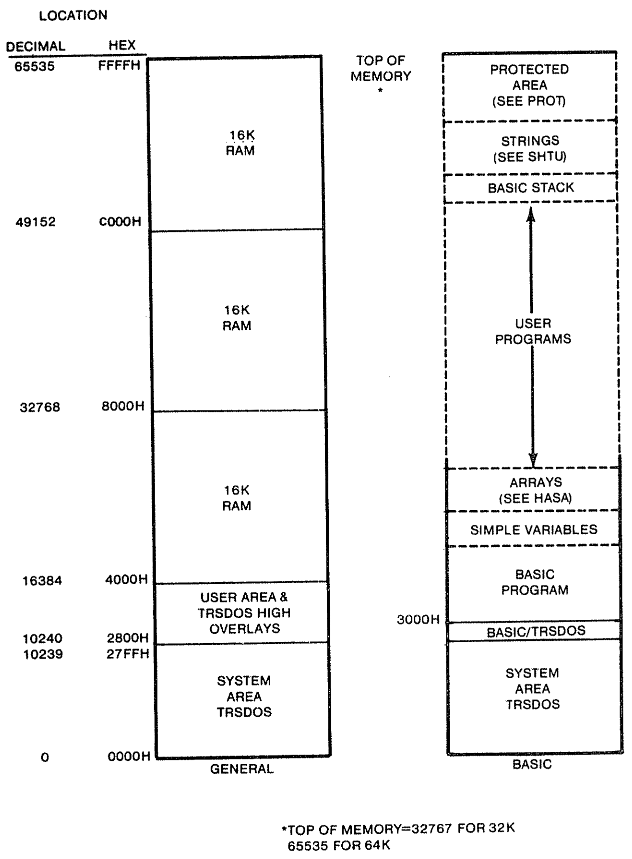
Figure MMM1-1 – *Memory Map, Model I/III*



MMM2
Memory Map, Model II

Figure MMM2-1 shows a memory map of the Model II.

Figure MMM2-1 – *Memory Map, Model II*



MOB2
MOD Operator in BASIC, Model II

The Model II MOD operator allows you to find the "modulus" result. What's a modulus? Glad you asked...

Do a divide of two numbers. The remainder is the modulus result. Example: 102/9=11, remainder 3. The value 3 is the "100 MOD 9" result. Not too handy, you say? Well, not that uncommon. One example: If a timer starts counting in minutes, the "minute hand" position on a clock can be given by

```
Elapsed Time in Minutes MOD 6Ø
```

MOER
MO Error

General catch-all for Missing Operand in a function, as in

```
1ØØ A=        'missing value here
```

MSDC
Multiple Serial Devices, Color Computer

Suppose that you have a serial line printer and a modem. How do you connect both of these to the Color Computer, which has only one serial output port (see BSCC)? Although some users have made a "Y" cable out of the four-wire serial cable and connected it to two devices, this doesn't always work (see Figure MSDC-1). A better solution for using one serial device at different times is to use a "4-pole double-throw switch" (see Figure MSDC-2) to connect either serial device 1 or serial device 2.

Figure MSDC-1 – *Color Computer Y Cable*

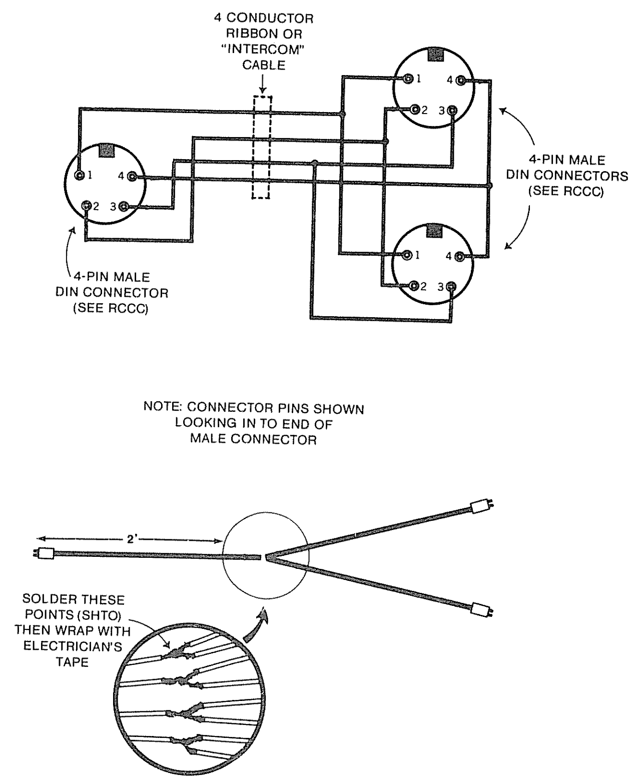
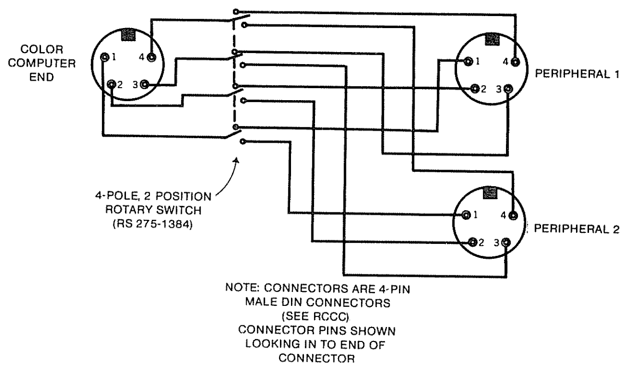

4 CONDUCTOR RIBBON OR "INTERCOM" CABLE
4-PIN MALE DIN CONNECTORS (SEE RCCC)
4-PIN MALE DIN CONNECTOR (SEE RCCC)
NOTE: CONNECTOR PINS SHOWN LOOKING IN TO END OF MALE CONNECTOR
2'
SOLDER THESE POINTS (SHTO) THEN WRAP WITH ELECTRICIAN'S TAPE

Figure MSDC-2 – *Color Computer Serial Switch*


COLOR COMPUTER END
PERIPHERAL 1
PERIPHERAL 2
4-POLE, 2 POSITION ROTARY SWITCH (RS 275-1384)
NOTE: CONNECTORS ARE 4-PIN MALE DIN CONNECTORS (SEE RCCC) CONNECTOR PINS SHOWN LOOKING IN TO END OF CONNECTOR

MSLH
Multiple Statement Lines in BASIC, How to Use, All Systems

Combine several statements in one BASIC line by separating the statements with colons. The code

```
1ØØ A=B*567+I
11Ø PRINT"A=";A
12Ø GOTO 1ØØØ
```

can be combined into the single line

```
1ØØ A=B*567+I:PRINT"A=";A:GOTO 1ØØØ
```

Are there any problems in this? Generally it's a good policy. It saves RAM by reducing the number of times the line number and associated "pointers" are stored (see LFBA) and saves execution time by making the BASIC lines easier to search. It's impossible to perform a GOTO or GOSUB to a command in the middle of another line, however.

The biggest kicker is this:

If you have an IF . . . THEN or IF . . . THEN . . . ELSE statement, you may get unexpected results. In an IF . . . THEN statement where the IF condition is true, then all of the multiple-statement line is executed. If the IF condition is false, then none of the remaining statements are executed.

```
1ØØ IF A=B THEN PRINT "MATCH": GOTO 1ØØØ
11Ø ...
```

If A = B in the above, "MATCH" will be printed and a jump would be made to line 1000. If A was not equal to B, line 110 would be executed. The IF . . . THEN . . . ELSE works in similar fashion; the remaining statements are executed only if the ELSE condition is in force.

MTOC
Motor, Turning On in BASIC, Color Computer

In BASIC, simply execute MOTOR ON or MOTOR OFF to turn the cassette on or off. You can use this command to control external devices by connecting the two pins that would normally go to the REMote jack on the cassette recorder as described in CCAM.

MTPB
Merging Programs, Disk BASIC

If you are using Disk BASIC on the Model I, II, III,or Color Computer, follow this procedure to merge two or more programs:

1. Determine the sequence of the programs — which one should be in the beginning, which is next, etc.

2. Load the first program. Renumber this program by following the procedure in RNBL (TRSDOS) or RNBM (LDOS). Note the number of the last line.

3. SAVE the program on disk as an ASCII file by using SAVE"name",A.

4. Load the second program. Renumber, using a starting line number larger than the last line number of the first program. SAVE the program as an ASCII file.

5. Load the remaining programs, renumber, and save as ASCII files, making certain that the starting line numbers are greater than the previous ending line numbers.

6. Load the first program by LOAD"name".

7. Merge the second program by

```
MERGE"name"
```

8. Merge the remaining programs by MERGE commands.

9. You now should have a larger program that consists of all the programs in sequence. Renumber if desired, and save as a new file by the SAVE command (the "A" option does not have to be used, unless you want an ASCII file).

Remember, MERGE is a true MERGE; if the line numbers of two or more programs are the same, the new line will overwrite the old line!

MUPF
Magazines, Using Programs from

In general, programs in magazines tend to be of lower quality than those found in books, or software that is sold. Magazines construct articles around advertising copy (you thought it was the other way around?) and tend to pay writers on the order of $1.23 per hour for their programming efforts. This ensures that much garbage is printed. In general, stay away from magazine software; if it is *that* good, it is probably for sale in machine-readable form.

MWAT
Modems, What Are They?

The word "modem" stands for "modulator/demodulator" — mean anything to you? I didn't think so. Computers and buzz words . . . A modem is a device to allow your TRS-80 to send data over telephone lines via the RS-232-C port (see RSWI and Figure MWAT-1). The "modulator" portion takes a digital input signal (either a one or a zero) and converts it into one of two audio tones. If the input is a constant 0, one tone is produced; if the input is a constant 1, a second tone is produced. If the input signal changes rapidly, as is the case with an RS-232-C "serial" signal, the audio output will alternate between one tone and the other and you'll hear a "blopety-blop-blopety-blop" type of output on the phone line.

At the other end of the telephone line, another modem takes the "blopety-blop-blopety-blop" high-speed switching between the two tones and converts back (demodulates) into signal levels of 0s and 1s.

And that's the primary function of a modem, to convert electrical signal levels from the RS-232-C port into audio tones that can be transmitted over Ma Bell. If your system is the system originating the call (as, for example, calling up CompuServe or a Bulletin Board system — see CPSU and BBUS), it is an "originate" modem and uses the frequencies 1270/1070 hz for transmitting data and 2225/2025 hz for receiving data. If your system answers the call (as, for example, the CompuServe computer system) it is an "answer" modem and uses the frequencies 2225/2025 hz for transmitting data and 1270/1070 hz for receiving data. In full-duplex operation (see RSWI), both systems can send data simultaneously and often do. An important point: some users think that if they have an originate only modem they cannot both transmit and receive data. Wrong! Originate only modems can simply not be used as an answering device with the answering frequencies. Most new modems allow you to switch between the originate and answer frequencies to select an "originate" or "answer" modem.

Along with the primary job of transmitting data, modems also have secondary functions. Signals relating to "carrier" (an audio tone from the other modem), "ring" (telephone ringing for an "auto-answer modem") and other functions are passed back to the RS-232-C port.

Strictly speaking, a modem connects the computer RS-232-C port directly to the phone line via a standard phone plug (see Figure MWAT-2), while an "acoustic coupler" connects the RS-232-C by a small microphone and speaker in "cups" of the coupler. The telephone handset is set into the two coupler cups, as shown in Figure MWAT-3. Modem now generally means both types. The direct-connect modem is more error-free, as it is an electrical link only.

"Auto-dial" modems dial a pulse or Touch-Tone number under program control. "Auto-answer" modems automatically answer the phone and initiate data communications. Everything can be under program control, and two systems can communicate without human intervention.

Figure MWAT-1 – *Modem Operation*

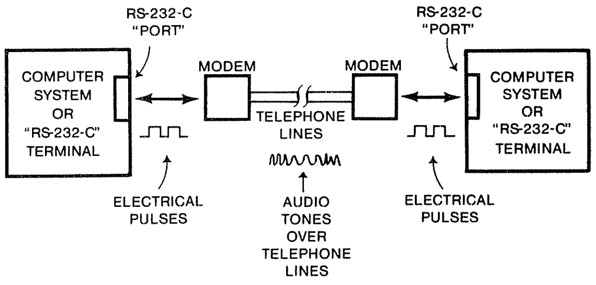

RS-232-C "PORT"
RS-232-C "PORT"
COMPUTER SYSTEM OR "RS-232-C" TERMINAL
MODEM
MODEM
COMPUTER SYSTEM OR "RS-232-C" TERMINAL
TELEPHONE LINES
ELECTRICAL PULSES
AUDIO TONES OVER TELEPHONE LINES
ELECTRICAL PULSES

Figure MWAT-2 – *Modem Connections*

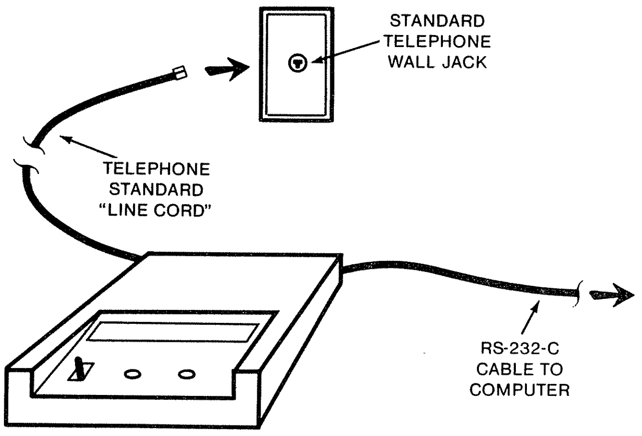
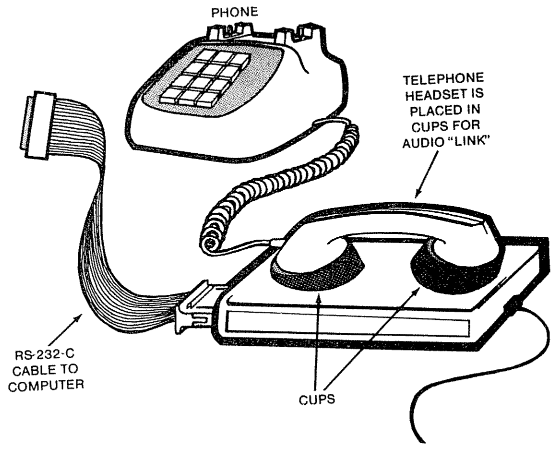

STANDARD TELEPHONE WALL JACK
TELEPHONE STANDARD "LINE CORD"
RS-232-C CABLE TO COMPUTER

Figure MWAT-3 – *Coupler Connection*


PHONE
TELEPHONE HEADSET IS PLACED IN CUPS FOR AUDIO "LINK"
RS-232-C CABLE TO COMPUTER
CUPS

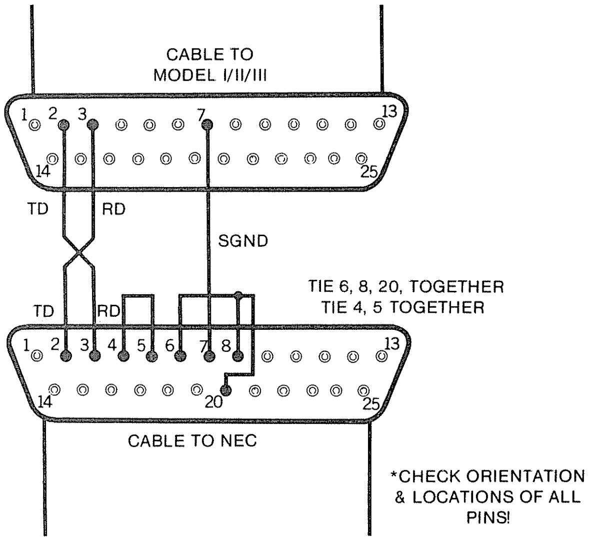
NECS
NEC Spinwriter, Operation Notes

For RS-232-C Spinwriters Only:

Connection: If you don't understand serial interfacing, then read RSWI. You'll need an RS-232-C "driver" program to communicate with the Spinwriter.

Most drivers will work with the cabling shown in Figure NECS-1 between the Spinwriter and Model I, II,or III.

Figure NECS-1 – *Connections for 300-Baud NEC Spinwriter*



Setup: There's no question about it, it's a chore to setup the switches on the Spinwriter. Refer to the NEC manual to find the location of the option switches. Set the NEC baud switches to 300 (ON and ON). Set the NEC SPEED switch to "M."

Setup your computer RS-232-C channel for 300 baud, 7 word bits, 2 stop bits, and no parity; setup the NEC switches similarly.

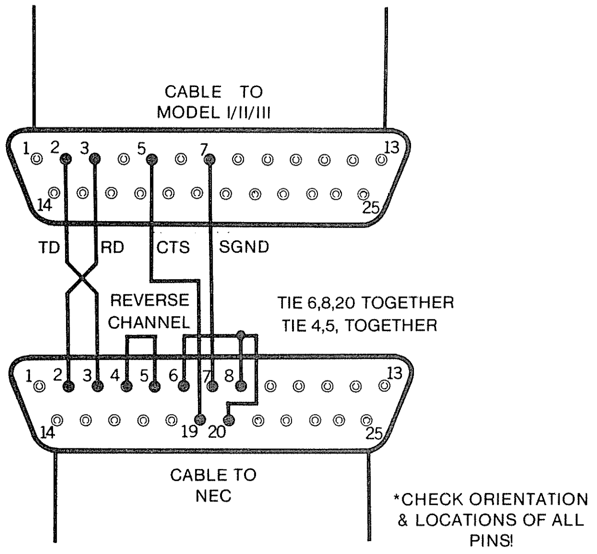
A data communications printer driver (such as LCOMM in LDOS) should now cause printing.

Try the 300 baud version to get your feet wet. If you want to operate at higher baud rates, then the "protocol" is somewhat different. The Spinwriter uses a buffer to accumulate serial characters. At higher speeds the printer can not keep pace with the computer output, and the buffer will overrun. The "REVERSE CHANNEL" provides an indication that the buffer is 7/8 full and is connected to the Clear to Send line on the computer side. Signal CTS must be checked by the computer terminal program as a "ready" condition.

Use the cabling shown in Figure NECS-2 to operate at baud rates up to 1200 baud.

Notes on Programming: The Spinwriter will print the standard ASCII character set (see ADFW) without problems, but also uses a set of "Escape Sequences" for special functions such as horizontal or vertical tabbing and many others. An escape sequence is a string of characters, the first of which is an Escape (ASCII 27). Here are some notes:

Figure NECS-2 – *Connections for 1200-Baud NEC Spinwriter*



To change vertical line spacing use the escape sequence 27, 93, and 80 through 95. The last character sets the vertical spacing (80 is 1/48", 81 is 2/48", etc).

```
100 FOR I=80 TO 95
110 A$=CHR$(27)+CHR$(93)+CHR$(I)+"HELLO"
120 LPRINT A$
130 NEXT I
```

To tab to any position, use the escape sequence 27, 80 through 85, and 64 through 95. Refer to chart in your NEC manual. Tab positions are 1 - 163; to tab to 95, do:

```
110 A$=CHR$(27)+CHR$(82)+CHR$(94)
120 LPRINT A$
```

To vertical tab, forward or back use 27, 88 though 91, and 64 through 95 (see chart in manual):

```
100 INPUT V
110 W=ABS(V)
120 S=INT(W/32)+88
130 T=W-INT(W/32)*32+64
140 IF V>=0 THEN S=S+2
150 LPRINT CHR$(27)+CHR$(S)+CHR$(T);
160 GOTO 100
```

To ring a bell, LPRINT CHR$(7).

To change horizontal spacing, use the escape sequence 27, 93, and 65 through 79. The last character changes the spacing in increments of 1/120" from 1/120" (65).

```
100 FOR I=65 TO 79
110 A$=CHR$(27)+CHR$(93)+CHR$(I)+"AB"
120 LPRINT A$
130 NEXT I
```

To print forward, output an "ESC >" (27,62). To print in reverse, output an "ESC <" (27,60). Note that a string of characters must be reversed on reversed printing:

```
100 LPRINT CHR$(27)+CHR$(62)+"ABCDEFG"
110 LPRINT CHR$(27)+CHR$(60)+" GFEDCBA"
```

To set margins, move to position by tabbing, and then output an "ESC M" for left margin or "ESC J" for right margin.

Proportional spacing is done in software by horizontal spacing as above, based on the width of each character.

See the NEC manual for other escape sequences.

NFER
NF Error

Next without FOR in XXXX. Your code has a NEXT I or similar statement that does not match a FOR I=XX TO XX STEP XX statement (see FTST). BASIC keeps track of each FOR I=XX TO XX statement and expects a NEXT I statement *somewhere*. Conversely, it does not expect to find a NEXT I if there is no initial FOR I=XX TO XX statement. If you have "nested" FOR. . .TO statements, check to see that there is one for every NEXT. Check also that you haven't transposed variable names, as in

```
100 FOR I=0 TO 100
110 FOR J=1 TO LEN(A$)
  .
  .
1000 NEXT I        'oh, oh! should be J
1100 NEXT J        'should be I
```

NRER
NR Error

No RESUME error. You're in the error-processing mode (initiated by an ON ERROR GOTO command, see ETIB), and the program has ended without a RESUME. Did you mean it to? If not, make certain that a RESUME is executed before program end. See ETIB.

NSWD
No System, What Does it Mean?

If you attempt to load TRSDOS and get a "NO SYSTEM" display, it means that you attempted to load a "data diskette" (see DDWA) without an operating system. TRSDOS and LDOS are too large to fit into memory, and must be loaded from disk. The disk you're using is good for storing data files, but doesn't contain the operating system modules.

ODER
OD Error

Out of Data Error

You've done too many READs for the data in the DATA list. DATA statements establish a data list (see DSRC). You may READ the list as long as you do not execute more READs than there are data items in the list. Common error — multiple READs that go past the end of the list as in:

```
100 DATA 1,2,3,4,5,6,7,8    '8 items
110 READ A,B,C              'read 1,2,3
120 READ D,E,F              'read 4,5,6
130 READ G,H,I              'read 7,8,...duh...
```

OGHU
ON . . . GOSUB, ON . . . GOTO, BASIC, All Systems

ON . . . GOSUB and ON . . . GOTO are basically the same, except that the ON . . . GOSUB calls a BASIC subroutine (see SUBB), saving the return statement location, while ON . . . GOTO simply transfers control to another part of the BASIC program and does not record the location of the statement after ON . . . GOTO.

The ON . . . GOSUB and ON . . . GOTO are called "computed" GOSUBs or GOTOs. The format for both is

```
ON (expression) GOXXX line#1, line#2,...line#n
```

The expression may be a numeric value (trivial case), a variable, or expression (such as A*B+3). The expression is first "evaluated." The expression is typically a value of 1 through some ending number. If the expression is equal to 1, a GOSUB or GOTO is then made to the line #1 line; if the expression is equal to 2, a GOSUB or GOTO is done to the line#2 line, and so forth. If A=3, for example, a GOSUB to 1036 would be done here

```
1000 ON A GOSUB 1020, 1022, 1036, 1040
```

If the expression is 0, or is greater than the number of line numbers, than the GOSUB or GOTO is not made and the next statement in sequence is executed. If A=0, or A was greater than 4, for example, statement 1010 would be executed here

```
1000 ON A GOSUB 1020, 1022, 1036, 1040
1010 ...
```

ON . . . GOSUBs and ON . . . GOTOs are used to "branch out" on to a number of subroutines or code segments based upon a single variable. A typical use might be a selection of a "menu" item list and a branch out to specific processing for the menu function.

OMER
OM Error

Out of Memory Error

You cheap somagun! You should have bought more than 16K of RAM!

If you've run out of memory, you'll have to compress your program (see CPM3), cut down on the size of your string area (see OSSH), cut down on the size of your arrays (see HASA), segment your program, or knuckle under to that RS salesman with 132 teeth who asks, "How May I Help You?"

ONIB
Octal Notation in BASIC, Some Systems

Octal notation is a shorthand form of binary.

Model I/III Disk BASIC, Model II BASIC, and Color Computer Extended Color and Disk BASIC allow octal notation. Hexadecimal would normally be used in lieu of octal, although some "fields" in machine language instructions might be suitable for octal representation.

Model I, II, III and Color Computer: Use the prefix "&O". A value of &O4000 is OCTAL 4000 or decimal 2,048. The value &O34 is octal 34 or decimal 28.

Model II: Use the BASIC OCT$ command to find the equivalent octal value of a number, as a string:

```
100 PRINT OCT$(X)
```

for example, will print the value of variable X as an octal string.

OSER
OS Error

Out of string space error. Allocate more string space by a CLEAR statement (see OSSH) or reduce the number of string variables (make them intermediate results, rather than variables). If you get an OM error on doing a new CLEAR, you're going to have to reduce the memory requirements (see CPM3).

OSSH
"Out Of String Space", What to Do About It

You haven't defined an area that BASIC can use for string manipulations, or you haven't defined a large enough area. Use a CLEAR command (Model I/III Level I excepted) at the start of your program to define a larger string "working storage" area. This would be a command similar to "100 CLEAR 2000"; in this case 2000 bytes are set aside.

How much storage is required? You could go through and laboriously calculate this, but no one does. If your BASIC program is small and you have 32K or 48K, set aside 3000 or 4000 bytes. If you have a large BASIC program, you'll have to trade off string working storage space with space used for the program, space used for variable storage, and so forth, see memory maps at MMM1, MMM2, and MMCC (I/III, II, and Color Computer, respectively). If you have a very large program, you may have to reduce your program size to fit everything into memory.

OVER
OV Error

Overflow Error

My cup runneth over! You've specified an integer, single-precision, or double-precision number that is too large for the range, as in

```
100 I%=32768           '+32767 max
110 A=34.56E99         'E99 too large
120 A#=34.555E99       'still too large
```

PCCC
PAINT Command, Extended BASIC, Color Computer

The PAINT command is well-named. Instead of drawing a figure, it PAINTs an existing figure, as shown in Figure PCCC-1. The command specifies a starting coordinate, a color to be painted, and a boundary color. The color to be painted fills up the entire area until the boundary color is encountered. PAINT is a convenient way to draw a figure with the "outline" commands such as LINE, CIRCLE, and DRAW, and then to fill in the figures with color.

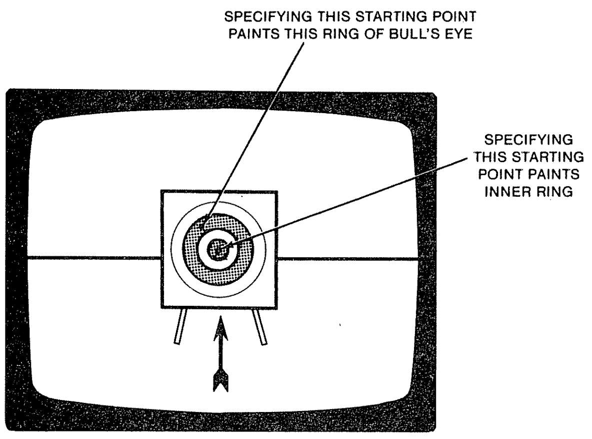
The format of PAINT is

```
100 PAINT (x,y),c,b
```

where x and y are screen coordinates (see CCPA), c is the color code (see GMIC), and b is the "boundary" color. The PAINT will take effect until the boundary is reached.

Warning: If there are "gaps" in the boundary, the paint will "leak out" into the surrounding screen until a new boundary is found, if any.

Figure PCCC-1 – *PAINT Command Use*


SPECIFYING THIS STARTING POINT PAINTS THIS RING OF BULL'S EYE
SPECIFYING THIS STARTING POINT PAINTS INNER RING

PDF1
PRINTing a Disk File, Model I TRSDOS

Use the PRINT command. It works just like LIST (see LDF1) except that the ASCII file is printed on the system line printer. The file must be an ASCII file as described in AFWA.

PHTU
Parentheses, How to Use

Parentheses group expressions in BASIC. The BASIC statement

```
100 A=B/2+5
```

is not the same as

```
100 A=B/(2+5)
```

There are rules that the BASIC interpreter uses for evaluating BASIC lines (and you thought it just went helter-skelter . . .). The rules are

1. BASIC scans the line from left to right.

2. BASIC looks for parentheses and evaluates the expressions inside the parentheses first.

3. If there are "nested" parentheses, BASIC works from the innermost set of parentheses out.

Okay, so far Now, assume that we're on the same level of parentheses, and BASIC is going to evaluate the operations. (If there are no parentheses, BASIC would now be at the line beginning.) The basic BASIC sequence is:

4. Process exponentiation first. Exponentiation raises a number or expression to a power and is indicated by an up arrow or left bracket (↑ or [ ).

5. Process signed numbers indicated by "-" or "+". These are "unary" operators as they require only one argument. Note that this is *not* addition or subtraction.

6. Process multiplication or division operators indicated by "*" or "/".

7. Process addition or subtraction operators as indicated by "+" or "-".

8. Process relational operators (<,>, <=, etc.).

9. Process logical operators (NOT, AND, OR).

A good way to remember a portion of this is by the mnemonic "My Dear Aunt Sally," standing for the "hierarchy" of multiplication, division, addition, and subtraction. Actually, there's more to the mnemonic, but some children may be reading this book . . .

Here's a simple example of the above:

```
1ØØ R=P*((I*(1+I)XN)/((1+I)XN-1))
```

BASIC would evaluate the expression as follows: (By the way, the expression is for finding the receipt or uniform series payment at interest rate I over N interest periods with principal P. With California "creative financing" in home purchases, and first, second, third, fourth, and fifth mortgages on homes, formulas such as these are indispensible to survival in real estate. "Can't make the deal? How much are you short? I think I have that in loose change . . Just sign this sixth mortgage paperwork. . . ").

```
1ØØ R=2ØØØØØ*((.Ø15*(1+.Ø15)↑36Ø)/(1+.Ø15)↑36Ø-1))
    '18% interest

11Ø R=2ØØØØØ*((.Ø15*(1.Ø15)↑36Ø)/(1.Ø15)↑36Ø-1))
    'innermost

12Ø R=2ØØØØØ*(.Ø15*212.694)/(212.694-1))
    'exponentiation

12Ø R=2ØØØØØ*(3.19Ø41/211.694)
    'innermost

12Ø R=2ØØØØØ*.Ø15Ø7Ø9
    'resolved

12Ø R=3Ø14.18
    'slum housing house pmt
```

PIEM
Period, in BASIC Edit Mode, Most Systems

Period (.) in BASIC edit mode stands for "current line," the last referenced BASIC line. For example,

```
>LIST31Ø
>EDIT.
 31Ø-
```

would display line 310 and then enter BASIC Edit mode (see EMBH) with line 310 setup for the edit. Not too handy.

(Does not apply to Color Computer.)

PLPB
PRINTs to LPRINTs and Back Again

Ever want to change PRINTs to LPRINTs after debugging a BASIC program? Or to change LPRINTs back to PRINTs for further debugging (after your brother-in-law ran the program for pork-belly futures and it blew up?). There a number of methods:

1. The most laborious: Manually go through the program and change between LPRINT and PRINT (or PRINT and PRINT#-2 for the Color Computer) by the Edit Mode. True, it is laborious, but works well for short programs. If you have a Model III, you can do the search easily by the CMD"X" command. CMD"X" searches for all occurrences of a reserved word or string literal. Figure PLPB-1 shows the dialogue in searching for the reserved word "PRINT"; searching for other reserved words or string literals is identical. Use double quotes around the string literal, and no quotes around the reserved words.

2. The search BASIC program in RAM method: BASIC "compresses" lines by using numeric values for BASIC keywords, called "tokens" (see TM13, TMTW, and TBCC). BASIC line format is shown in LFBA. Each line ends with a 0 byte. The first two bytes of the line are a pointer to the next line in binary. The next two bytes are the current line number, in binary. The last BASIC program line has a next line pointer of 0. Valid BASIC tokens are non-ASCII and have values of 128-255; all other bytes are ASCII characters.

The starting address of the first BASIC line is kept in a system pointer, dependent upon the system. Sounds like everything we need . . .

Figure PLPB-1 – *CMD'X' Use in Model III*

```
                 THIS IS NOT ENCLOSED IN QUOTES,
                 THEREFORE IT IS A RESERVED
                 WORD (TOKEN, SEE TM13).
                  /
>CMD"X",[PRINT]
1ØØ    13Ø    14Ø    3ØØ    41Ø    1143    12ØØ }  LINE NOS.
132Ø   1413   2ØØØ   2567                      }  AT WHICH
READY                                             PRINT TOKEN
>                                                 FOUND
```

Figure PLPB-2 shows a short BASIC program that will search a BASIC program in RAM (including itself) for a given token and change these tokens to another token value. It will start at a given line number and end at a given line number. You can use this to change PRINTs to LPRINTs or back again, or to change any token to any other token, although its not immediately apparent why you would want to change all GOTOs to ELSEs . . .

Refer to TM13 (I/III), TMTW (II) or TBCC (Color Computer) to get token values. Enter the starting and ending line numbers as required.

Warning: If you enter the incorrect token values, or modify the PLPB-1 program itself with incorrect values, you may destroy your BASIC program. As with all of these programs, I'll disavow any knowledge of your actions if you do . . .

Figure PLPB-2 – *Token Changer from Memory*

```
100 ' TOKEN CHANGER
110 INPUT "TOKEN TO SEARCH FOR";ST
120 INPUT "NEW TOKEN";NT
130 INPUT "START LINE #";SL
140 INPUT "END LINE #";EL
150 PN=PEEK(16548)+PEEK(16549)*256
160 WP=PN: PN=PEEK(WP)+PEEK(WP+1)*256: LN=PEEK(WP+2)+PEEK(WP+3)*256
170 WP=WP+4
180 IF PN=0 THEN GOTO 220
190 IF LN<SL THEN GOTO 160 ELSE IF LN>EL THEN GOTO 160
200 IF PEEK(WP)=0 THEN GOTO 160 ELSE IF PEEK(WP)=ST THEN POKE WP,NT
210 WP=WP+1: GOTO 200
220 STOP

                         NOTE: MODEL I/III ONLY
                               USE WP-65536 FORM
                               OVER 32K
```

POCE
Pseudo-Ops, Using, Color Computer EDTASM+ Assembler

Pseudo-ops stands for "pseudo-operations," assembler commands that do not generate machine language instructions, but instruct the assembler in other actions. The pseudo-ops described here are for the Color Computer EDTASM+.

ORG: The ORG pseudo-op sets the ORiGin of the following code. This is normally the first non-remark source line in an assembly language source file. If no ORG is used, the assembly "location counter" will be set to 0 and all following code will be assembled from location 0. Unless all code is relocatable, an ORG must be used.

Multiple ORGs reset the assembler location counter. Normally this would be to a higher location. Here's a sample of a program designed to run at RAM location **$3FOOH**:

```
        ORG     $3F00H     set origin
START   LDX     NEXTT      load buffer address
         .
         .
         .
NEXTT   EQU     *          start of text buffer
        ORG     *+200      200 bytes in buffer
```

In this case the assembler assembled code that ran from **$3F00** on through to NEXTT. At NEXTT, the ORiGin was set to the "current location counter" (* +200). In effect, this bypasses 200 bytes on the object module load.

If you are using "in-memory" assembly (see EAIM), the ORG will not be necessary, as the assembler will automatically "relocate" the code to the proper in-memory area. An ORG may be put into the final version which will go on cassette and be loaded from BASIC by the CLOADM command (see LEMC). However, you can use the ORG even in in-memory assembly provided that the /MO, manual origin, is specified. See EAIM.

END: Normally used as last line in source code. If argument omitted, start of BASIC is used. If you do not use an argument, you'll have to remember the start address after the CLOADM (see LEMC). If you do use an argument, you can simply do an EXEC without an address after the CLOADM.

```
END             load and go to BASIC
END     START   load, go to program start
```

EQU: The EQU equates a label to an argument. Typically this is used for mnemonic purposes, to equate an easily-remembered name to a numeric value, or to assign multiple names to a location. Sample:

```
CR      EQU   $ØD         ASCII CR
START   EQU   *           start of program
HASHP   EQU   *           HASHP same as START
        LDB   CR          same as LDA $ØD
```

FCB: Form Constant Byte (These mnemonics were all made up by an ex-Army man"Form Constant Byte, Harrrrchhh!"). Generates a single byte at the current assembler location. One argument only, making one DEFB for every constant byte (groan!) necessary.

```
TABLE   FCB   65          generate ASCII "A"
        FCB   13          generate $ØD
        FCB   $13         generate 19 decimal
```

FDB: Form Double Byte (Hut!) Generates a double byte (word) at the current assembler location. One argument only. Note: Data is generated in standard 6809E address format, most significant byte, followed by least significant byte.

```
TABLE   FDB   $123A       generates $12, $3A
        FDB   START       generates start address
```

RMB: Reserve Memory Bytes. Doesn't generate data at all. Simply advances assembler location counter by number of bytes in argument. On load, the area defined by the RMB and argument will be bypassed without any data being loaded. Used to allocate storage, similar to DIM in BASIC.

```
TABLE   RMB   1ØØØ        1ØØØ bytes, program fills
```

FCC: Form Constant Character. Generates an ASCII text string, similar to A$="xxxxx" in BASIC. One ASCII byte generated for each character in argument. The format is

```
        FCC     /THIS IS A STRING/
```

where the slash is the "delimiter," computerese for a character marking the beginning or end of a field. The delimiter may be any character that will not be used in the string.

```
MSG1    FCC     'ENTER ID NUMBER'   generates 15 chars
```

SET: Seldom used. Allows labels to be redefined similar to EQU.

```
CR      SET     $ØD             carriage return 1

CR      SET     $0A             carriage return 2
```

SETDP: Set Direct Page. This pseudo-op instructs the assembler to set the direct page register. In the direct mode of addressing, the 6809E forms an address by an 8-bit address in the instruction (least significant byte) plus an 8-bit address from the DP register (most significant byte). To properly assemble data in this mode, the assembler must be instructed as to the value in DP.

The program normally would also contain a TFR A,DP or similar instruction to setup the direct page. The argument may be $00 through $FF, for 256 pages of 256 bytes each.

```
SETDP   $Ø3         set DP to $Ø3 for $Ø3XX
LDA     $Ø3         $Ø3XX
TFR     A,DP        setup DP in program
```

Figure POCE-1 illustrates these pseudo-ops.

Figure POCE-1 – *6809E Assembler Pseudo-Ops*

```
ØØ1ØØ * EDTASM+ PSEUDO OPS
3FØØ            ØØ11Ø         ORG    $3FØØ    SET ORIGIN
         ØØ17   ØØ12Ø POSN    EQU    23       POSN EQUATED TO 23
         ØØØ3   ØØ13Ø DEST    SET    3        WILL BE REDEFINED
3FØØ     41     ØØ14Ø TABLE   FCB    65       GENERATE ONE BYTE
3FØ1     65     ØØ15Ø         FCB    $65      ONE BYTE IN HEX
3FØ2     3FØØ   ØØ16Ø         FDB    TABLE    GENERATE TWO BYTES
3FØ4     1234   ØØ17Ø         FDB    $1234    TWO BYTES
3FØ6            ØØ18Ø         RMB    1ØØ      RESERVE 1ØØ BYTES
3F6A     54     ØØ19Ø         FCC    /THIS IS A STRING/
         48
         49
         53
         2Ø
         49
         53
         2Ø
         41
         2Ø
         53
         54
         52
         49
         4E
```

```
             47
             ØØØ4      ØØ2ØØ DEST    SET      4       REDEFINE
             Ø3        ØØ21Ø         SETDP    $Ø3     SET DP TO $Ø3XX
3F84                   ØØ22Ø         ORG      *+1Ø    RESERVE 1Ø BYTES
3F84         17        ØØ23Ø         FCB      POSN    USE EQUATE
             ØØØØ      ØØ24Ø         END
ØØØØØ TOTAL ERRORS

DEST    ØØØ4    S
POSN    ØØ17
TABLE   3FØØ
```

POZA

Pseudo-Ops, Using, Z-80 Assemblers, Models I, II, III

Pseudo-op stands for "pseudo-operation", an assembler command that does not generate machine language instructions, but instructs the assembler in other actions. The pseudo-ops described here are for the RS Series I/III EDTASM, the RS earlier cassette-based EDTASM, the NEWDOS EDTASM, and EDAS from MISOSYS. Differences will be noted.

ORG: The ORG pseudo-op sets the ORiGin of the following code. This is normally the first non- remark source line in an assembly language source file. If no ORG is used, the assembly "location counter" will be set to 0 and all following code will be assembled from location 0. Unless all code is relocatable an ORG must be used.

Multiple ORGs reset the assembler location counter, Normally this would be to a higher location. Here's a sample of a program designed to run at RAM location **8000H**:

```
        ORG     8ØØØH       ;set origin
START   LD      HL,NEXTT    ;load buffer address
        .
        .
        .
NEXTT   EQU     $           ;start of text buffer
        ORG     $+2ØØ       ;2ØØ bytes in buffer
```

In this case the assembler assembled code that ran from **8000H** on through to NEXTT. At NEXTT, the ORiGin was set to the "current location counter" ($+200). In effect, this bypasses 200 bytes on the object module load.

END: Normally used as last line in source code. If argument omitted, start of DOS or BASIC (non-disk) is used.

```
END             ;load and go to DOS
END     START   ;load, go to program start
```

EQU: The EQU equates a label to an argument. Typically this is used for mnemonic purposes, to equate an easily-remembered name to a numeric value, or to assign multiple names to a location. Sample:

```
CR      EQU     ODH         ;ASCII CR
START   EQU     $           ;start of program
HASHP   EQU     $           ;HASHP same as START
        LD      A,CR        ;same as LD  A,ØDH
```

DEFB: Generates a single byte at the current assembler location. One argument only, making one DEFB for every constant byte (groan!) necessary (at least on Radio Shack assemblers).

```
TABLE   DEFB    'A'         ;generate ASCII "A"
        DEFB    13          ;generate ØDH
        DEFB    13H         ;generate 19 decimal
```

DEFW: Generates a double byte (word) at the current assembler location. One argument only. Note: Data is generated in standard Z-80 address format, least significant byte, followed by most significant byte (see Z8AF).

```
TABLE   DEFW    123AH       ;generates 3AH, 12H
        DEFW    START       ;generates start address
```

DEFS: Doesn't generate data at all. Simply advances assembler location counter by number of bytes in argument. On load, the area defined by the DEFS and argument will be bypassed without any data being loaded. Used to allocate storage, similar to DIM in BASIC.

```
TABLE   DEFS    1ØØØ        ;1ØØØ bytes, program fills
```

DEFM: Generates an ASCII text string, similar to A$="xxxxx" in BASIC. One ASCII byte generated for each character in argument.

```
MSG1    DEFM    'ENTER ID NUMBER'  ;generates 15 chars
```

DEFL: Seldom used. Allows label to be redefined. Similar to EQU.

```
CR      DEFL    ØDH         ;carriage return 1
        .
        .
CR      DEFL    OAH         ;carriage return 2
```

***LIST OFF**: Not really a pseudo-op. Turns off program listing at point encountered plus one line.

***LIST ON**: Turns on program listing at point encountered.

Figure POZA-1 illustrates these pseudo-ops.

Figure POZA-1 – *Z-80 Assembler Pseudo-Ops*

```
                    00100 ; Z-80 PSEUDO OPS
8000                00110          ORG     8000H    ;ORIGIN AT 8000H
0001                00120 ESCAPE   DEFL    1        ;WILL BE REDEFINED
C000                00130 START    EQU     0C000H   ;EQUATE "START" SYMBOL
8000 20             00140 TABLE    DEFB    32       ;GENERATES ONE BYTE
8001 32             00150          DEFB    32H      ;ONE BYTE IN HEX
8002 00C0           00160          DEFW    START    ;GENERATES ONE WORD
8004 4D             00170          DEFM    'MESSAGE'
     45.53 53 41 47 45.
0001                00180 ESCAPE   DEFL    1        ;REDEFINED
C000                00190          ORG     0C000H   ;REDEFINE ORIGIN
C000 18FC           00200          DEFW    -1000    ;GENERATES ONE WORD
0064                00210          DEFS    100      ;RESERVES 100 BYTES
                    00220 *LIST OFF                 ;TURNS OFF LISTING
                    00280 *LIST ON                  ;TURNS ON LISTING
0000                00290          END
00000 Total errors
```

PPKU

PEEk and POKE, Using, Model I/III/Color Computer

PEEK and POKE are used:

- To change certain system variables, such as Device Control Blocks (POKE).
- To look at ROM to see what's happening (PEEK).
- To read parameters from machine language programs after having called the program (PEEK).
- To store parameters before calling a machine language program (POKE).
- To interface to certain system devices, such as line printers or RS-232-C hardware from BASIC.
- To implement high-speed graphics displays by POKEing into video memory.

Both PEEK and POKE work with a memory address. The memory address on all systems is a number between 0 and 65,535. Depending upon the system, this 64K (64*1024) bytes of memory may be divided up into ROM, RAM, or input/output addresses (see MMM1 or MMCC for the I/III and Color Computer, respectively). You should know where you are POKEing, although you can *usually* PEEK with impunity.

The format of PEEK is PEEK(XXXXX) where XXXXX is the address of a byte in memory. A

```
100 A=PEEK(I)
```

will set variable A equal to an 8-bit value from the memory location specified in I. Specifying PEEK(32767), for example, reads back the contents of RAM memory location 32767 (**&H7FFF**). The 8-bit value will be from 0 through 255. (It will be 255 if the memory location doesn't exist.)

If you are PEEKing into locations above 32767, and have a Model I/III you will have to use the PEEK form PEEK(I-65536) to fool the BASIC interpreter into thinking that the argument is a valid integer argument. An unfortunate condition, but necessary; I is the memory address in this case.

The format of POKE is POKE I,D, where I is the memory address (use I-65536 for locations above 32767 on a Model I or III), and D is the data value to be POKEed. The data value must be a number from 0 through 255. The value will be stored in the memory location specified, or sent to an input/output device. (You cannot store in Read-Only Memory, or ROM!)

PROT
Protecting Memory in BASIC, Models I/II/III and Color Computer

To protect memory:

On powering up or at any time on the Color Computer: "CLEAR XXX,MMMMM." The area above MMMMM will not be used. The value XXX is the number of bytes that will be allocated as "string storage area" and is typically 200 bytes; allocate more if you will be doing a great deal of string processing. The area above MMMMM can now be used for any purpose . . . without the BASIC interpreter, or other system programs, using it for system functions. CLEAR 200,&H3EFF – this example allocates 200 bytes for string storage and protects the area from **3F00** through **3FFF** on a 16K machine (to the top of memory on a larger machine).

On powering up or TRSDOS load on the Model I/III: For the MEMORY SIZE? prompt, reply with one less than the first location to be used; the area above this location will not be used by the BASIC interpreter or by any other system program for any system function. Replying with MEMORY SIZE? 31999, for example, will protect memory location 32000 on up to "top of memory." Top of memory for a Model I/III 16K machine is 32767 (**7FFF**), for a 32K machine is 49,151 (**BFFF**) and for a 48K machine is 65,535 (**FFFF**). Replying with 31999 for a 32K machine, for example, would protect RAM locations 32000 through 32767. Simply pressing ENTER after MEMORY SIZE does not protect memory.

Protected memory is normally used for storage of user machine language programs, but can also be used for special data areas.

Model I/III LDOS: When LBASIC is loaded from disk, use this form of the command file load to protect memory:

```
LBASIC (MEM=XXXXX)
```

where XXXXX is one less than the first location to be protected. To load BASIC and protect from location 32000 on, for example, enter

```
LBASIC (MEM=32000)
```

```
to load LBASIC.
```

Model II: To protect memory use this form of the BASIC load:

```
BASIC -M:31999
```

This example protects memory from 32000 up.

PRUU
PRINT USING, BASIC, Using

(Not applicable to Model I/III, Level I and Color Computer BASIC).

PRINT USING is used to "format" data for printing or display. It solves the problems of columnating, right justification of columns, rounding off cents, and adding zeroes to fractional amounts. LPRINT USING works the same as the following description except that it uses the system line printer instead of the display.

The format of PRINT USING is

```
PRINT USING string;item list
```

where "string" is a PRINT USING string defining the format and "item list" is a list of variables or strings to be printed. A typical PRINT USING string to print dollar amounts up to $999.99 might be A$="$$ ##.##" and the PRINT USING statements might be

```
100 A$="$$##.##"
110 PRINT USING A$;BAL 'print balance
```

The PRINT USING string may be a string constant, defined in the PRINT USING statement, or may be a string variable. Here's how to do a variety of things with PRINT USING:

1. To print a fixed-length numeric field:

   A. Positive variables: Use the " #" specifier. A " #" defines one character in the field. If you know that your amounts will be 0 through 999 and want a fixed-length string for columnating, do

```
100 PRINT USING "###";A
```

   to get outputs of bb0, bb5, b23, 666, and 999, where b is a blank. Note that no "leading zeroes" will be printed, but that blanks will be printed in their place.

   Use a " #" for every position you want printed; if you need values up to 999,999, for example, use " ######."

   B. Positive and negative variables: Use the " #" specifier as in 1A. Positive values will be printed out as above; negative values will be printed out with a leading negative sign.

```
100 PRINT USING "####";A
```

   will produce outputs of b345, 9999, bb-1, or -999. Note that you must allow another " #" to get the full range of negative digits.

2. To print a mixed number with decimal point, use the "decimal point" specifier and " #". Putting a decimal point in the PRINT USING string results in an automatic printout of the decimal point and all digits specified for the fraction.

```
100 PRINT USING "###.###";A
```

will result in printouts of bb5.452, b79.899, 999.451, b-2.558, -34.677, and -99.999. The string printed will always be seven characters long.

P

3. To print a leading or trailing sign: Use a "+" specifier. This specifier means "print a positive or negative sign" at the print position.

```
100 PRINT USING "+###.###";A
```

for example, will produce values of b+34.444, +999.999, bb-1.123, and -999.999. A "+" at the end (as used in some accounting applications) will generate a trailing sign *if negative* but not if positive - results like 999.999b and 999.999-.

4. To print a dollar sign with leading blanks, use the "floating dollar sign" specifier.

```
100 PRINT USING "$$##.##";A
```

would produce values of $123.67, bb$6.77, and b$34.62, for a fixed-length string of 7 characters.

5. To print a comma, as between thousands, use a comma anywhere, just as a period is used for the decimal point.

```
100 PRINT USING "$$##,###.##";A
```

would include a floating dollar sign and produce values such as bb$4,456.67, b$45,999.50, and $999,999.99.

6. To print leading asterisks, dollar sign, commas, and decimal point for checks:

```
100 PRINT USING "**$##,###.##";A
```

would generate values such as *****$999.99, ***$9,999.99, **$99,999.99, and *$999,999.99.

7. To print exponential format, use the up arrow specifier (prints as a left bracket in some cases) for every character to be printed; normally this would be an "E" followed by a sign, followed by two digits.

```
100 PRINT USING "###.## ↑↑↑↑";A
```

would produce values such as bb3.44E+16, b56.91E-17, and 999.99E25.

8. To print the first character of a string: Use the "!" specifier. The "item" in this case is a string variable.

```
100 PRINT USING "!";A$
```

would produce "W" if A$ was "WILLIAM" and "T" if A$ was "TERRY."

9. To print the first "n" characters of a string, use the "%" specifier around intermediate blanks for every character of the string to be printed. The "item" in this case is a string variable.

```
100 PRINT USING "% %";A$
```

would produce "123" if A$ was "12345.67" and "BIL" if A$ was "BILL BARDEN, JR."

Have we just about exhausted PRINT USING, or is the reverse true? Here are some additional rules:

1. If the item list consists of more than one item, then all items in the list will use the same format, as defined by the PRINT USING string:

```
100 PRINT USING "###.##";A, B, C
```

2. Field specifiers may be strung together in the PRINT USING string. Characters other than PRINT USING specifiers will print out.

```
100 PRINT USING "%  %--%  %--%%";A$, B$, C$
```

for example, will print out "ABCD—GHIJ—XY" if A$="ABCDEF", B$="GHIJKLMNOPQRSTUVW", and C$="XYZ". (I'm sure you can find hundreds of applications for this example . . .)

3. The PRINT USING statement may not work if you attempt to confuse the BASIC interpreter by such strings as " ###. ##, ##.##". After all, a computer is only human.

PSBC
PSET, PRESET, and PPOINT in BASIC, Color Computer

PSET, PRESET, and PPOINT are PMODE SETs, RESETs, and POINTs. They do essentially the same thing as the Color BASIC commands - set, reset, or test a point. Of course, they do it with finer resolution, depending upon the mode.

Read CCPA to find out about the graphics resolution.

The format for setting a POINT is

```
PSET (x,y,c)
```

where x and y are the coordinates (see CCPA) and c is the color code (see GMIC).

The format for resetting a point is

```
PRESET (x,y)
```

where x and y are the coordinates (see CCPA). The color after the reset will be the current background color.

The format for PPOINT is the same as PRESET:

```
PPOINT (x,y)
```

The color code (see GMIC) is returned as an argument. For example, in

```
100 A=PPOINT(100,100)
```

variable A will be set to the color code of point 100,100.

PSWU
Power Supplies, What to Use

Need a power supply for a computer interface application?

Color Computer only: If your interface is the joystick port, +5VDC is available on pin 5 (see DICC). Current requirements should be limited to a few milliamperes. The CC also has -12V @ 100 ma, +12V @ 300 ma, and +5V @ 300 ma on pins 1, 2, and 9 on the ROM cartridge connectors. Ground is pin 33 or 34. See RCCN.

Alternative 1: Batteries. If your circuit draws tens of milliamperes, but not hundreds, it is feasible to use a battery power supply. If you require + 5 volts, however, do *not* use a 6 volt battery; it will burn out many integrated circuits. Use batteries for generating RS-232-C voltages, power to CMOS, or linear integrated circuits, or other places where the voltage does not have to be precise. Generally, the larger the battery, the longer it will last. Also, believe it or not, the more expensive the battery, the more cost effective it is (based on my own testing of Radio Shack batteries).

Typical life for a Radio Shack 9V 23-553 battery, 360 milliampere-hours; that's 20 hours for current of 18 milliamps. Use larger batteries for greater current requirements.

Alternative 2: +5 Volt Supply with Battery. If you use the circuit shown in Figure PSWU-1, you can develop a precise +5 volts for most digital logic. The battery must supply more than about +6.5 volts. "Heat-sink" the 7805 regulator if it runs too hot. (It may be too hot too comfortably hold a finger on and still operate effectively.)

Alternative 3: Other supplies with Battery. Use the same circuit shown in Figure PSWU-1 with the 7812 and 7815 regulators to develop voltages of 12 and 15 volts.

Alternative 4: Use a low-cost +9 or +12 volt supply. Radio Shack currently sells a selectable 4.5, 6, 7.5, or 9 volt battery eliminator that will supply 300 milliamps for $9.95 (RS 270-1551). Use directly, or with a voltage regulator as shown in Figure PSWU-1. Power supplies that are "car battery eliminators" are also available from the Shack and supply about 2 amps at about $30. Again, use the 7805 or other regulators, but heat sink them very well, as shown in Figure PSWU-2.

Figure PSWU-1 – *Battery Operation*

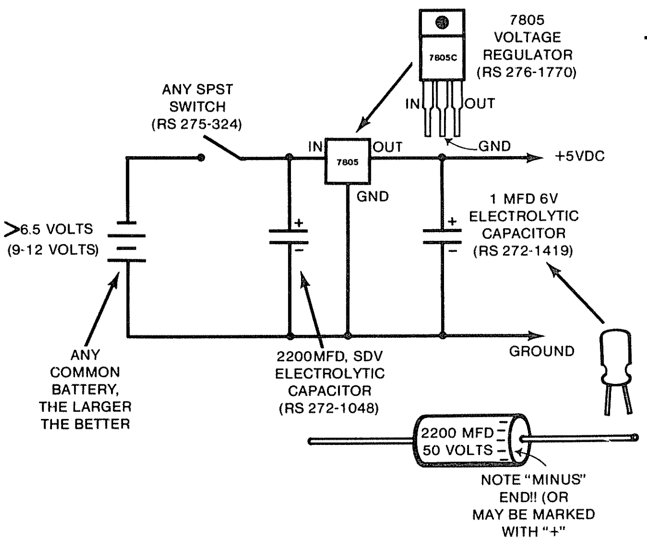

7805C
7805 VOLTAGE REGULATOR (RS 276-1770)
IN
OUT
ANY SPST SWITCH (RS 275-324)
IN
7805
OUT
GND
GND
+5VDC
1 MFD 6V ELECTROLYTIC CAPACITOR (RS 272-1419)
>6.5 VOLTS (9-12 VOLTS)
GROUND
ANY COMMON BATTERY, THE LARGER THE BETTER
2200MFD, SDV ELECTROLYTIC CAPACITOR (RS 272-1048)
2200 MFD 50 VOLTS
NOTE "MINUS" END!! (OR MAY BE MARKED WITH "+"

Figure PSWU-2 – *Heat Sinking a Regulator*

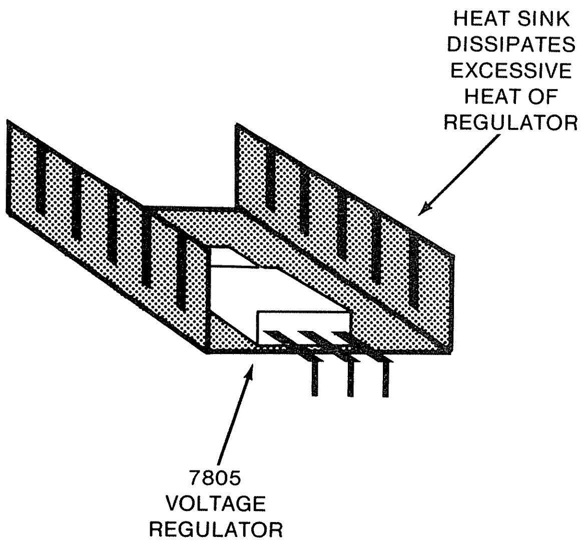
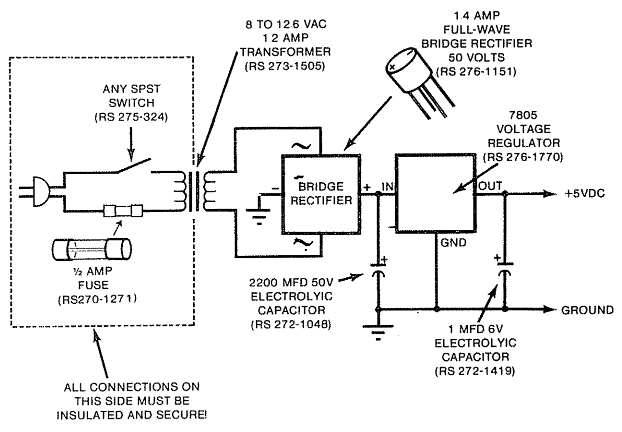


Figure PSWU-3 – *Power Supply Design*



Alternative 5: I can see you like to "roll your own" . . . If you have some kit building experience, build the circuit shown in Figure PSWU-3. It's almost foolproof and works as well as the above power supplies. Use a 12.6 volt, 1.2 amp transformer (RS 273-1505) or a 3-amp version. It will provide up to 1.5 amperes output when the moon is in the proper phase or 1 amp with no problems at all. Heat sink the 7805 regulator properly, as shown in Figure PSWU-2.

BEWARE

110 VAC can kill! Fuse the circuit as shown, solder all connections, insulate all exposed leads, and use a secure cover for the power supply.

PTSC
Printing the Screen, Model I/II/III TRSDOS/LDOS

See DSPR to "dump" the contents of the screen to the system line printer if in BASIC.

Model I/III TRSDOS:

If you want to duplicate what is displayed on the screer to the line printer, enter

```
DUAL (ON)      (DUAL ON for Model II)
```

to a TRSDOS READY. Everything that is displayed on the screen will be printed on the system line printer. (Well, just about everything. Some Radio Shack system programs, such as the F function in DEBUG, and other programs that use their own input/output will not duplicate the screen).

Enter

```
DUAL (OFF)
```

to disable the printer (DUAL OFF for Model II).

Model I/III LDOS:

Execute the LDOS command

```
LINK *DO *PR
```

This command "links" the Display Output Device (normally screen) to the Print Device (normally printer) so that anything sent to the screen is sent to the printer as well. Reset the linkage by a

```
RESET *DO
```

PVM1
Passing Variables to Machine Language Programs, Model I or III

If you are interfacing BASIC to a machine language program or programs (see CML1 or CMD1 for information on general interfacing concepts), you may want to pass "parameters" back and forth. An example of this is a machine language subroutine that takes the time and number of the day of the year and converts it into elapsed seconds. The parameters passed *to* the subroutine would be current time and the "Julian" day, the number of the day of the year from 1 to 366. The parameter passed *back* would be the total number of elapsed seconds. (Never mind that this is a dumb example. It's not easy writing a book like this and being creative . . .)

How can the parameters be passed?

If no parameters are to be passed, then the USR call or USRn call simply uses a "dummy" return variable and a dummy argument as in

```
1ØØ A=USRØ(Ø)
```

where variable A is the dummy return variable and (0) is the dummy argument. The machine language subroutine will be called and executed, but no data will be passed between BASIC and the machine-language program.

If a single parameter is to be sent *to* the subroutine, but no parameter is to be returned (as in the case of a subroutine to translate and print one character) then the argument in the USR call can be a "real" argument. However, the argument must be an integer argument with a value of -32,768 through +32,767. Suppose we wanted to pass a value of 247 to the machine language subroutine. We'd have something like

```
1ØØ A=247
11Ø B=USRØ(A)
```

Where would the argument be when we entered the subroutine? Somewhere in BASIC. One of the very first things the subroutine would have to do would be to perform a CALL 0A7FH instruction. This action would load the argument from the USR call into the HL register pair. From that point on, the machine language code could proceed with processing of the argument in HL.

If a parameter is to be sent back *from* the subroutine, then the very last thing the subroutine should do is to perform a JP 0A9AH instruction. This instruction will take the contents of the HL register and convert it to the BASIC variable specified in the USR call. Since the HL register must hold a 16-bit value, the variable will be within integer limits.

What about passing a parameter to and from a subroutine? Perform a CALL#0A7FH upon entry and a JP 0A9AH on exit from the subroutine. Sixteen-bit parameters will be passed to and from the subroutine.

What about *multiple arguments*? The USR call mechanism only allows one 16-bit integer argument to and from the subroutine. If more then one argument is to be passed then we've got to get clever. Pack two 8-bit arguments by code such as

```
100 A=123*256+45.   'first arg=123, second=45.
110 B=USR(A)        'call subroutine
```

This can be extended into four 4-bit arguments, and so forth.

If you have many arguments to be passed, you must go to an *argument list* concept. In this method the arguments to be passed to and from the subroutine are either in a predefined area of memory or a *pointer* to an argument list is passed. Suppose that an X, Y, and Z value were to be passed to a subroutine. We could define the argument area at locations **8000H**, **8001H**, and **8002H**, POKE them before calling the subroutine, and then do the USR call. Alternatively, we could POKE the arguments somewhere else in RAM and then pass a 16-bit pointer to them by the USR call, as in

```
100 POKE 36864-65536,X     'X to 9000H
110 POKE 36865-65536,Y     'Y to 9001H
120 POKE 36866-65536,Z     'Z to 9002H
130 A=USR0(36864-65536)    'call SUBR 1st set
140 POKE 36867-65536,X     'X to 9003H
150 POKE 36868-65536,Y     'Y to 9004H
160 POKE 36869-65536,Z     'Z to 9005H
170 A=USR0(36867-65536)    'call SUBR 2nd set
```

Remember that any memory address over 32767 must have the form XXXXX-65536 to make the value acceptable to BASIC, as we've used here.

Arguments can also be POKEd into dummy strings and arrays, but this can get somewhat messy, as string locations and arrays are *dynamically* moved in RAM as BASIC code is edited, new variables are added, etc. Use the VARPTR function carefully if you take this tack, and make certain that the VARPTR directly precedes the USR call without introducing a new variable name.

A sample machine language program that searches for a given character on a given screen line is shown in Figure PVM1-1. If found, the character position on the line is returned. This program uses the two calls to ROM to pick up and pass back the arguments as shown in the figure.

Figure PVM1-1A – *Variable Passing Example*

```
100 DATA 205,127,10,124,38,0,6,6,41,16,253,1,0,60,9,229
110 DATA 6,64,190,40,9,35,16,250,225,33,255,255,24,4
120 DATA 193,183,237,66,195,154,10
130 FOR I=32768 TO 32768+36
140 READ A: POKE I-65536,A
150 NEXT I
160 DEFUSR0=32768-65536
170 INPUT"CHARACTER TO SEARCH FOR";A$
180 INPUT"SCREEN LINE 0-15";S
190 B=USR0(ASC(A$)*256+S)
200 IF B=-1 THEN PRINT "NOT FOUND" ELSE PRINT "FOUND AT CHARACTER POSITION";B
210 GOTO 170
```

```
                  ØØ1ØØ ;*******************************************************
                  ØØ11Ø ;* SCAN SCREEN LINE FOR CHARACTER                      *
                  ØØ12Ø ;*    ENTRY: (H)=CHARACTER                             *
                  ØØ13Ø ;*           (L)=LINE #, Ø-15                          *
                  ØØ14Ø ;*    EXIT:  (HL)=CHARACTER POSITION (Ø-15) IF FOUND   *
                  ØØ15Ø ;*           OR -1 IF NOT FOUND                        *
                  ØØ16Ø ;*******************************************************
ØØØØ CD7FØA       ØØ17Ø FNDCHR   CALL   ØA7FH       ;get character, line #
ØØØ3 7C           ØØ18Ø          LD     A,H         ;save character
ØØØ4 26ØØ         ØØ19Ø          LD     H,Ø         ;line # now in HL
ØØØ6 Ø6Ø6         ØØ2ØØ          LD     B,6         ;6 counts
ØØØ8 29           ØØ21Ø FNDØ1Ø   ADD    HL,HL            ;*2
ØØØ9 1ØFD         ØØ22Ø          DJNZ   FNDØ1Ø           ;line #*64
ØØØB Ø1ØØ3C       ØØ23Ø          LD     BC,3CØØH    ;start of video
ØØØE Ø9           ØØ24Ø          ADD    HL,BC       ;point to line start
ØØØF E5           ØØ25Ø          PUSH   HL          ;save start
ØØ1Ø Ø64Ø         ØØ26Ø          LD     B,64        ;64 characters in line
ØØ12 BE           ØØ27Ø FNDØ2Ø   CP     (HL)             ;test for character
ØØ13 28Ø9         ØØ28Ø          JR     Z,FNDØ8Ø         ;go if found
ØØ15 23           ØØ29Ø          INC    HL               ;bump pointer
ØØ16 1ØFA         ØØ3ØØ          DJNZ   FNDØ2Ø           ;loop if not found
ØØ18 E1           ØØ31Ø          POP    HL          ;reset stack
ØØ19 21FFFF       ØØ32Ø          LD     HL,-1       ;not found flag
ØØ1C 18Ø4         ØØ33Ø          JR     FNDØ9Ø      ;return
ØØ1E C1           ØØ34Ø FNDØ8Ø   POP    BC          ;start of line
ØØ1F B7           ØØ35Ø          OR     A           ;reset carry
ØØ2Ø ED42         ØØ36Ø          SBC    HL,BC       ;find char position
ØØ22 C39AØA       ØØ37Ø FNDØ9Ø   JP     ØA9AH       ;pass char position back
ØØØØ              ØØ38Ø          END
ØØØØØ Total Errors
```

PVMC
Passing Variables to Machine Language Programs, Model II or Color Computer

If you are interfacing BASIC to a machine language program or programs (see CMLC or CMEC for information on general interfacing concepts), you may want to pass "parameters" back and forth. For example, you might want to move a block of data from one location in memory to another by a machine language subroutine, passing the starting and ending locations of two blocks, for a total of four parameters.

If no parameters are to be passed, then the USR call or USRn call simply uses a "dummy" return variable and a dummy argument as in

```
1ØØ A=USRØ(Ø)
```

where variable A is the dummy return variable and (0) is the dummy argument. The machine language subroutine will be called and executed, but no data will be passed between BASIC and the machine language program.

Unfortunately, both the Color Computer and the Model II have a cumbersome way of passing arguments, passing a pointer to the argument rather than the argument itself. The best way to pass arguments to and from a machine language subroutine is to go to an *argument list* concept. In this method the arguments to be passed to and from the subroutine are either in a predefined area of memory or a *pointer* to an argument list is passed. Suppose that an X, Y, and Z value were to be passed to a subroutine. Using Color Computer RAM memory space as an example, we could define the argument area at locations **$3F00**, **$3F01** and **$3F02** (16128, 16129 and 16130 decimal) POKE the arguments before calling the subroutine, and then do the USR call. Alternatively, we could POKE the arguments somewhere else in RAM and then pass a 16-bit pointer to them by the USR call, as in

```
1ØØ POKE &H3FØØ,X        'set x
11Ø POKE &H3FØ1,Y        'set y
12Ø POKE &H3FØ2,Z        'set z
13Ø A=USRØ(&H3FØØ)       'call subroutine
14Ø POKE &H3FØ3,X        'set x
15Ø POKE &H3FØ4,Y        'set y
16Ø POKE &H3FØ5,Z        'set z
17Ø A=USRØ(&H3FØ3)       'call for second set
```

Arguments can also be POKEd into dummy strings and arrays, but this can get somewhat messy, as string locations and arrays are *dynamically* moved in RAM as BASIC code is edited, new variables are added, etc. Use the VARPTR function carefully if you take this tack, and make certain that the VARPTR directly precedes the USR call without introducing a new variable name.

A sample machine language program that searches for a given character on a given screen line is shown in Figure PVMC-1. If found, the character position on the line is returned. This program uses the predefined area to pick up and pass back the arguments as shown in the figure.

Figure PVMC-1A – *Variable Passing Example*

```
                  *********************************************
                  * SCAN SCREEN LINE                          *
                  *      ENTRY:  ($3FØØ)=LINE #, Ø-15.        *
                  *              ($3FØ1)=CHARACTER FOR SEARCH *
                  *              $3FØ2,3 RESERVED             *
                  *      EXIT:   ($3FØ2)=CHARACTER POS OR -1  *
                  *********************************************
ØØØØ B6   3FØØ    SCNLNE  LDA    $3FØØ   LINE #
ØØØ3 C6   2Ø              LDB    #32     # CPS PER LINE
ØØØ5 3D                   MUL            LINE#*32
ØØØ6 C3   Ø4ØØ            ADDD   #$4ØØ   START OF TEXT
ØØØ9 FD   3FØ2            STD    $3FØ2   SAVE LINE START
ØØØC 1F   Ø1              TFR    D,X     NOW IN X
ØØØE B6   3FØ1            LDA    $3FØ1   GET CHARACTER
ØØ11 C6   2Ø              LDB    #32     32 CHARACTERS IN LINE
ØØ13 A1   8Ø      SCNØ1Ø  CMPA   ,X+       TEST
ØØ15 27   Ø7              BEQ    SCNØ2Ø    GO IF FOUND
ØØ17 5A                   DECB             DECR COUNT
ØØ18 26   F9              BNE    SCNØ1Ø    GO IF NOT 32
ØØ1A C6   FF              LDB    #$ØFF   FLAG FOR NOT FOUND
ØØ1C 2Ø   Ø7              BRA    SCNØ9Ø  GO TO STORE
ØØ1E 3Ø   1F      SCNØ2Ø  LEAX   -1,X    ADJUST FOR AUTO INC
ØØ2Ø 1F   1Ø              TFR    X,D     PNTR TO D
ØØ22 B3   3FØ2            SUBD   $3FØ2   SUBTRACT LINE START
ØØ25 F7   3FØ2    SCNØ9Ø  STB    $3FØ2   STORE FLAG OR CP
ØØ28 39                   RTS            RETURN
          ØØØØ            END
ØØØØØ TOTAL ERRORS

SCNØ1Ø  ØØ13
SCNØ2Ø  ØØ1E
SCNØ9Ø  ØØ25
SCNLNE  ØØØØ
```

```
100 DATA 182,63,0,198,32,61,195,4,0,253,63,2
110 DATA 31,1,182,63,1,198,32,161,128,39,7,90
120 DATA 38,249,198,255,32,7,48,31,31,16,179
130 DATA 63,2,247,63,2,57
140 FOR I=&H3E00 TO &H3E00+40
150 READ A: POKE I,A
160 NEXT I
170 DEFUSR0=&H3E00
180 INPUT "LINE # FOR SEARCH";L
190 INPUT "CHARACTER FOR SEARCH";A$
200 POKE &H3F00,L: POKE &H3F01,ASC(A$)
210 A=USR0(0)
220 A=PEEK(&H3F02)
230 IF A=255 THEN PRINT"NOT FOUND" ELSE PRINT "FOUND AT";A
240 GOTO 180
```

PWDS
Passwords, Diskette, Model I/II/III

There are three distinct passwords used on all Radio Shack Model I/II/III diskettes and files, the master password, the access password and the update password.

Every diskette has a master password. The idea of the master password is to protect the diskette from unauthorized backup or modification of critical files. The "default" password for the master password is "PASSWORD." If you are assigning a master password to a diskette, use this name as the password if you don't want to specify another. When should you specify another? If you have total access to your system, then there's no need to specify a secret diskette password. If several people are using the system, use a secret master password if you're afraid of tampering. The master password is specified during FORMAT of the diskette (BDSM).

The master password is associated with the entire disk and must be specified during BACKUP (BDSM), PROT (ADCT, ADC2), or PURGE (DADF) operations. The other two passwords, access and update, are associated with each separate file on your diskette. They're created at the same time the disk file is created. For example, you might write a BASIC program and call it INVENT/NEW.TANDY, giving it the name INVENT/NEW and the password "TANDY". (See FNMH for file name formats.) From 1 to 8 characters can be used for a file name password, the first of which must be an alphabetic character. If you don't specify a password during file creation, 8 blank spaces are used for the password.

Actually, although it isn't obvious, there are *two* passwords created along with the file, the access and update passwords. Initially, they are both set to the same characters that you've specified, or blanks.

You can change either or both passwords by using the ATTRIB command (ADFC, ADFL) in TRSDOS or LDOS.

The update password allows complete access to a file. Anybody that knows the update password can KILL, maim, or do anything to the file.

The access password is used when you want to give other people limited access to the file, but not complete control. You might want to give your brother-in-law the ability to read and execute a BASIC inventory file, but not to write out to it. Changing the access password by ATTRIB allows you to do just that. (Never did trust the summagun . . .).

The access password makes it possible to protect a file so that it may be executed but not listed (so your brother-in-law can't see the code) or any other level of protection up to complete access.

For formats and use of ATTRIB see ADFC and ADFL.

PWII
PAUSE, What is It? TRSDOS/LDOS

PAUSE is a Model II/III TRSDOS command and //PAUSE is an LDOS JCL command. They're used in DO/JCL files (see JCLW) to pause execution and notify an Operator to take some action. Pressing ENTER will continue the DO file execution. PAUSE is like a BASIC comment line.

QWII
Quote, in BASIC, What Is It?

If a BASIC line starts with a single quote character, the entire line is a REMark line. The single quote replaces a colon, REM so that you may have multiple statement lines such as

```
100 A=C/2+56.7   'THIS IS ACTUALLY A SECOND STATEMENT
```

A BASIC REM command is used to start a remark line, a line that is ignored in any BASIC program but will LIST or print. Any number of REM lines can be used, and they can be put anywhere in the BASIC program.

notes

RADF
RENAMEing a Disk File TRSDOS/LDOS, All Systems

To change the name of a Disk File, use the RENAME command:

```
RENAME name1 TO name2
```

Both the name1 and name2 file names are in standard file name format (see FNMH). If the name1 file has a password associated with it, it must be part of the file specification. After the RENAME the file attributes including the password(s) (see PWDS) remain the same; only the name is changed to protect the innocent.

Naturally, you can't rename a file to an existing name.

LDOS Users: If the extension is not specified in name2, the new file name will use the name1 extension as the new extension.

```
RENAME ACCTS/FEB:Ø TO OLDMAS
```

will rename to OLDMAS/FEB. Use a slash without an extension to create a new file name without an extension

```
RENAME ACCTS/FEB:Ø TO OLDMAS/
```

Color Computer: Use quotes around the file names

```
RENAME "ACCTS/FEB" TO "OLDMAS"
```

RBWI
Reset Button, Where Is It?

Model I: Left rear of CPU (keyboard) case, just inside the cutout edge. Model II: Clearly marked "RESET" under power switch. Model III: Orange button near upper right corner of keyboard. Color Computer: Right rear of case, about 3 inches in.

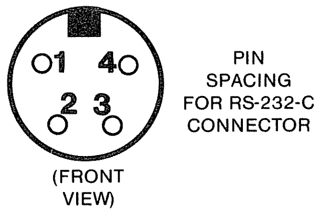
RCAD
Regaining Control of an AUTO Disk, TRSDOS/LDOS

Help! My computer is out of control! Under certain conditions, an AUTO disk will load and hang (* option used in LDOS and AUTO'ed program hangs or BREAK key disabled in other systems). To reset the AUTO in this case, load another diskette of the same system type, replace the second diskette with the AUTO diskette, and execute AUTO. The AUTO command will be reset on the diskette.

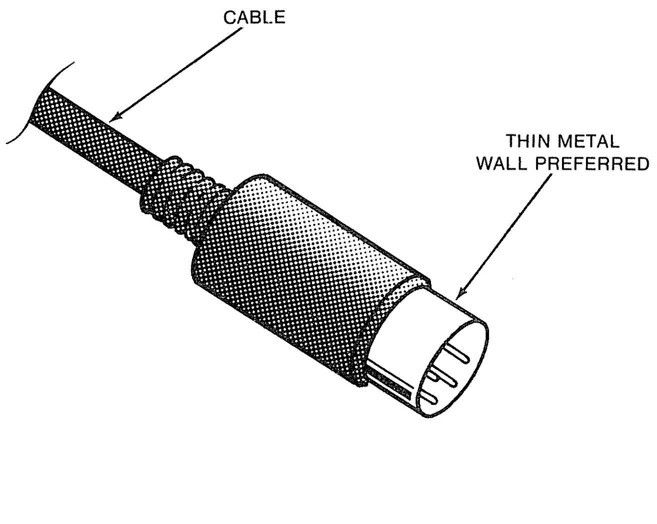
RCCC
RS-232-C Connector, Color Computer

The CC uses an odd 4-pin male DIN plug. DIN is a generic name for plugs that are similar to Figure RCCC-1. Radio Shack doesn't carry these currently, so you may be stuck buying the entire cable (RS 26-3020).

Figure RCCC-1 – *DIN Plugs*



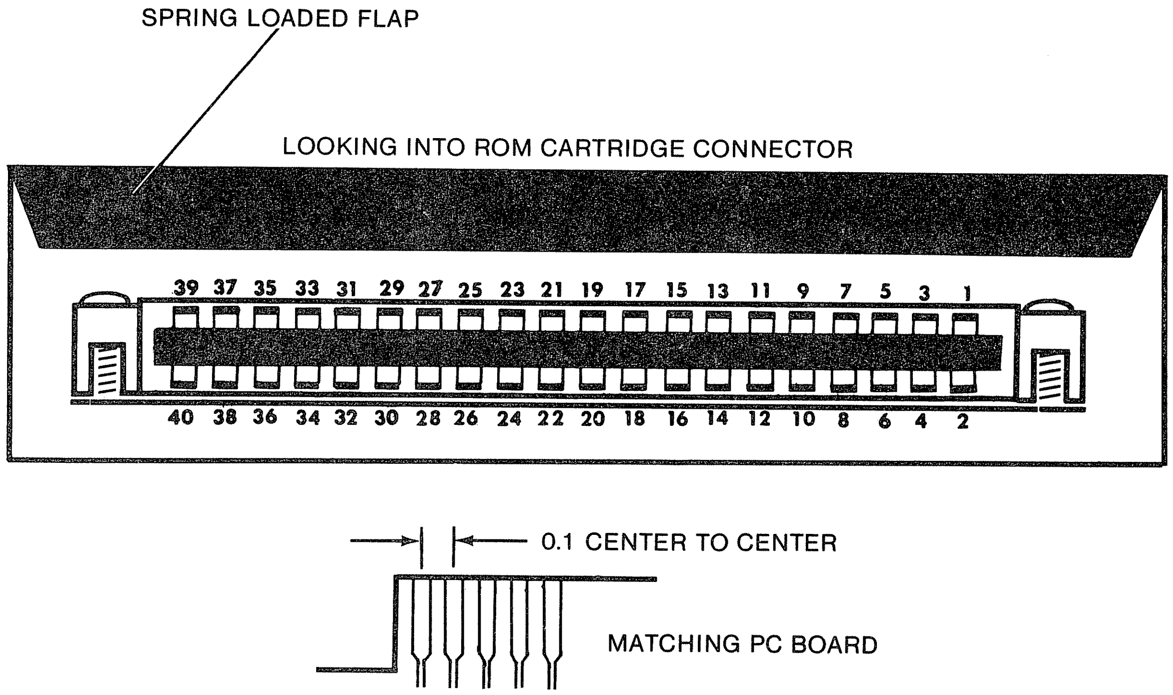
This plug is manufactured and available from other sources, however. Make certain that the walls of the plug are thin metal, otherwise the plug may not fit into the CC jack. Pin spacing looks as shown in the figure.



RCCN
ROM Cartridge Connector, Color Computer

Refer to Figure RCCN-1 to see the pin numbers of the ROM cartridge connector. A printed circuit board with pin spacing of 0.1 inches between centers will fit this connector. Pin numbers correspond to those in the Color Computer Technical Manual.

Figure RCCN-1 – *ROM Cartridge Connector Pinout*



RCWA
ROM Calls, What Are They? All Systems

A ROM call is a USR call that calls a machine-language subroutine. The USR call is primarily meant for calls to a machine-language program created by the user. In the case of ROM calls, however, a link is made to a subroutine in ROM (read-only-memory) or in other cases to system "code" in RAM.

The ROM contains the BASIC interpreter program, a large program that handles all processing and execution of BASIC programs. To accomplish this processing, the BASIC code in ROM uses many functional units — subroutines to read a character, to output to the line printer, to find a string, and so forth. The ROM calls take advantage of this existing code to make it easy for the user to perform certain functions.

If TRSDOS or LDOS is being used, there are additional subroutines available related to Disk file manage and DOS operations, along with BASIC "extensions."

ROM calls, and calls to system RAM subroutines, therefore, are used to simplify coding and to allow the user to take advantage of existing code.

All ROM calls (we'll use the term "ROM call" to mean both ROM and RAM calls) use a USR command in BASIC to cause the BASIC program to transfer control to the subroutine. Read PVM1 OR PVMC to see how the USR function works.

In some cases the ROM calls require "passing arguments" or parameters. Parameters may be passed to the ROM subroutine or from the ROM subroutine, or both. Read RCWA or PVM1 to get background on this aspect of ROM calls.

There are basically two types of ROM calls:

1. Those documented by "the Shack." These work from poor to excellent.

2. Those not officially supported by Radio Shack documentation. These work from poor to excellent.

ROM calls documented by Radio Shack generally work well (all kidding aside) and they will probably be redocumented in new ROM versions. Calls not officially supported by Radio Shack may be defined in one ROM version, but may be invalid in subsequent versions.

Documented ROM calls are such calls as "Read A Cassette Byte," "Dump Screen to Printer" and "Set RS-232-C." For both documented and undocumented calls, refer to IJG's excellent and comprehensive book "Microsoft BASIC Decoded" by James Farvour (Model I). This book shows a "disassembly" of all of ROM BASIC!

To make ROM calls, refer to the individual descriptions which will show you how to make the USR call and pass parameters. Common ROM calls are listed in Table RCWA-1.

Table RCWA-1 – *ROM Calls*

Models I and III:

Location	Description	Registers
33H	Display Character	Character to be displayed in A
1C9H	Clear screen	None
2BH	Keyboard Scan	Keyboard character in A on return or Ø if none
49H	Get keyboard character	Keyboard character in A on return
Ø3BH	Printer output	Character to be output in A
212H	Cassette select	A=Ø on entry to select cassette
287H	Write cassette leader and sync	None
264H	Write one cassette byte	Byte to be written in A on entry
296H	Read one cassette	Byte returned in A
1F8H	Turn off cassette	None

RDFB
Random Disk Files, BASIC, Using

Random disk files are one of the 2 types of files (the other is "sequential," see SDFB) that Disk BASIC can use. Read SDFB if you're not familiar with disk buffers and OPENing and CLOSEing disk files.

Reasons for Using Random Files: Random files are different from sequential files because they allow the user to access each record of the file directly, without reading in the entire file. (You might want to read SDFB to see how this is done in sequential files.) If you want the 100th record of a random file, you can simply say "GET 1,100" and BASIC will read in the 100th record alone. In the sequential file case, you'd have to read in all 99 preceding records before the 100th was found, and this is very time consuming.

The main reason for use of random files, then, is speed of access from disk.

Another reason for using random files given in RS documentation is that they "take up less space." In the general case, this probably isn't a true statement. Random files are oriented towards "fixed-" length records, whereas sequential files are variable length, depending upon the character length of the data and the line. Typical records in the random file case would be fixed at 128 bytes in length, and all records would have to conform to that 128-byte length. Suppose that you had 128 bytes describing a person's name and telephone number. "Ed, 555-1212" would have to be padded out to 128 bytes by blanks, and would take up just as much space as "Institute for the Study of Microcomputer Manufacturers and Other Shell Games, 555-9786."

Disk Sectors, Buffers, and Records: Random files are geared to disk sectors (see DSHL), all of which on the TRS-80 systems are 256 bytes in length. Although records in random files can be any length from 1 through 256 (0 is 256), it's probably somewhat more efficient to use lengths that are submultiples of 256 - 128, 64, and so forth. Use record sizes that are geared to your application, however. If you have an inventory file with a 6-digit part number, a 30-character description, a 6-digit supplier code, a 6-digit number on hand, and a 6-digit number on order, you would probably want to use a 64-byte record length to handle the 54 bytes of data.

ASCII Data in Random Files: Random files, like sequential files, are oriented towards ASCII data (see AFWA). Data is stored on the random files as ASCII characters. Numeric values are converted to ASCII and put into the file, and converted back from ASCII when taken out of the file.

General Flow for Creating a Random File: To create a random file, do this:

1. Execute a BASIC OPEN statement. The format of OPEN is

```
1ØØ OPEN "R",1,"INVENTOR/APR"
```

The above example "opens" a random file called INVENTOR/APR. The OPEN action doesn't write any records, but simply enters the name of the file in the disk directory, specs it as a random file and finds a vacant spot on the disk to use. At this point the DOS doesn't know how big the file is going to be. You might, however, have CREATEd disk space (see CDFH) or preallocated space on the disk, but this is not at all mandatory.

The buffer number is 1 through 15 and specifies a buffer area in RAM to be used with the file.

The record length in the above case is 256 bytes. To specify other sizes from 1 to 255, use

```
100 OPEN "R",1,"INVENTOR/APR",64
```

or a similar statement. You may have to specify a "V" during the BASIC sign-on for the number of buffers, also (see HMF1). The "V" stands for "variable-length" records, A complete misnomer. It really stands for "record sizes other than the standard, 256".

2. You've now OPENed the file and can write to it. Next, you have to "field" the buffer. This is a fancy term for defining how the 256 or 100 or 64 bytes of the *record* should be divided. Fielding is done (believe or not), by the FIELD statement. Taking that inventory example, again, let's field a 64-byte record for inventory records:

```
110 FIELD 1,6 AS PN$,30 AS DS$, 6 AS SC$, 6 AS OH$,
    6 AS OO$, 10 AS DUMMY$
```

The FIELD statement above specifies the first 6 characters as a part number string variable, the next 30 characters as a description variable, the next 6 characters as a supplier code variable, the next 6 characters as an on hand variable, the next 6 characters as an on order variable, and the last 10 characters as a "dummy," or padding. The "dummy" here wasn't necessary, but could be used as the first or intermediate field.

The first value after the FIELD is the buffer number associated with the OPEN statement.

Nothing mysterious here — all we've done is to specify what "fields" the record should be divided into. Each field has a name and length. The field names are similar but not identical to string variable names; they don't take up string space and are really mnemonic devices to remind you where the component parts of the record are.

3. You've now OPENed the file and specified where the fields of each record are located in the record. The next step is to store the data into the current record. The current record, by the way, is always in the disk buffer. If the record size is 256 bytes, it *is* the current buffer. If the record size is 64 bytes, it is 1/4 of the disk buffer.

There are two ways to store string variables into the record: by LSET and RSET. Taking the DS$ field, or description field, as an example, let's assume that the part description is in variable A$ and is "ARCHER LOW-LIFE BATTERY." This string is only 24 bytes in length, so how does it fit into a 30-byte field? Should there be 6 leading blanks or 6 trailing blanks? If you use

```
120 LSET DS$=A$
```

you'll "left justify" and have trailing blanks. If you use

```
120 RSET DS$=A$
```

you'll "right justify" and have leading blanks. It's up to you, and that's the purpose of LSET and RSET, to give you the option.

What about numeric variables? You could convert from a numeric variable to a string variable by STR$ (see CSNV), but there are three BASIC functions that are geared specifically to random buffer operations, MKI$, MKS$, and MKD$. These three functions convert an integer variable, single-precision variable, or double-precision variable to a string. To set the PN$, SC$, OH$ and OO$ fields in the current record, you'd use something like:

```
130 PN$=MKS$(CP)      'convert current part#
140 SC$=MKS$(CS)      'convert supplier code
150 OH$=MKS$(CH)      'convert on hand #
160 OO$=MKS$(CO)      'convert on order #
```

The lines above would do the conversion of the single-precision variables and put the results in the record as character (ASCII) data.

4. You've now OPENed the file, defined the fields, and stored data in the first record. To write this record to the disk file, do:

```
170 PUT 1,I
```

The PUT statement writes out the current record from buffer 1 to the Ith record in the disk file. Naturally, you'd have to change I as you went along. In this first case it would probably be 1, but you could write to the 100th record initially by

```
170 PUT 1,100
```

The file manage of TRSDOS or LDOS would automatically find the first or 100th record *position* in the disk file and write out the record to it.

If you have specified 64-byte records and listen for the disk writes during a routine such as this, you'll notice that a write occurs after every 4th record, if you're writing records in sequence. The disk buffer holds 4 records, and writes in this case are done after the buffer fills up with 4 records.

5. The steps above, filling the record with data, followed by a PUT, are done as many times as you'd like. Each time the PUT is done, *you* determine the next record to be written, based on your own scheme of sequencing. If you had part numbers from 1 to 100 and they were input out of sequence, you might write to the disk as PUTs to 54,56,3,45,77,1, and so forth, based on the current part number.

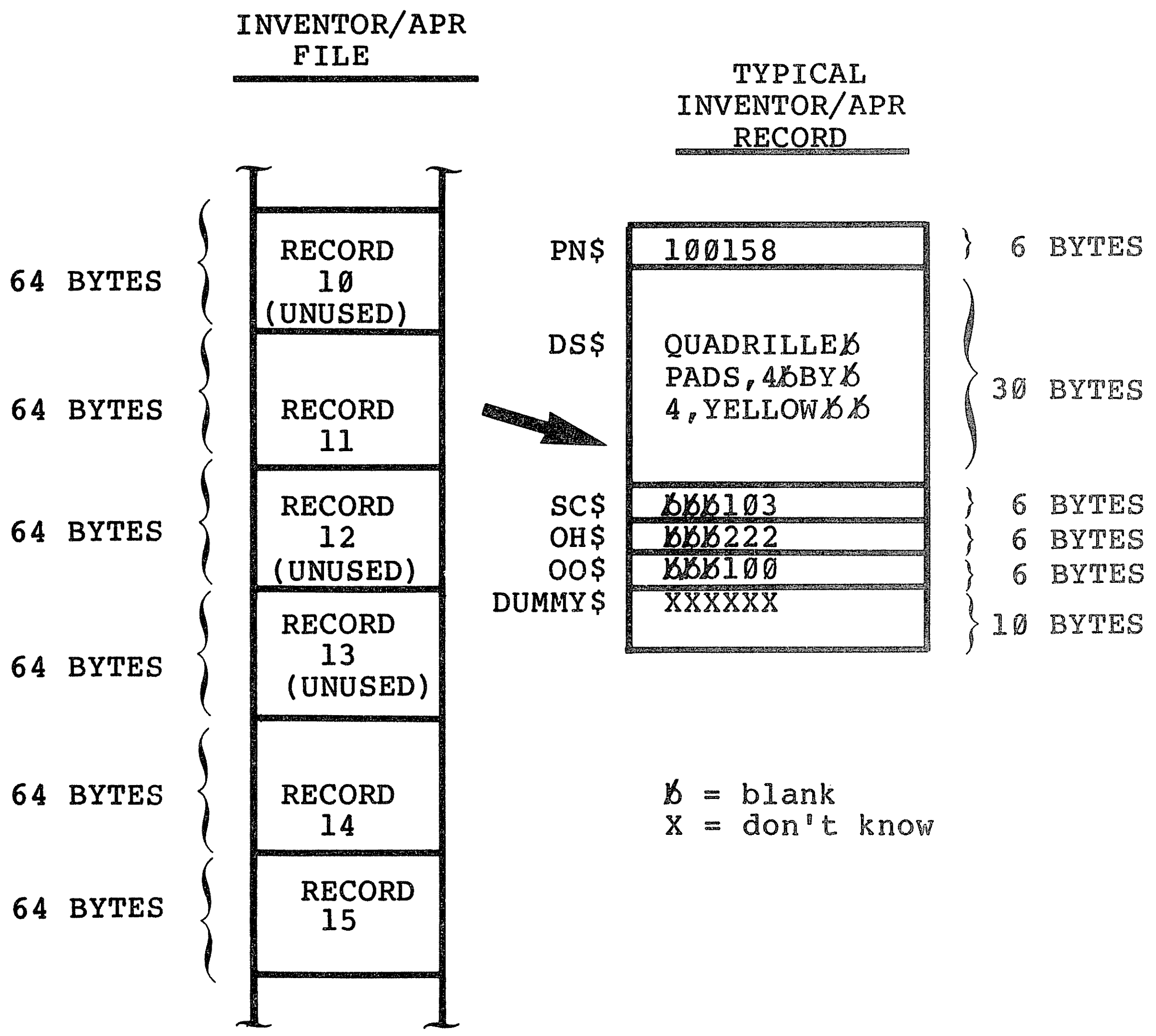
6. At the end of the data processing, you should CLOSE the file. CLOSEing the file "flushes" the buffer of any remaining data and properly terminates the directory entry. You'd have something like:

```
200 CLOSE 1
```

to close the file associated with buffer 1.

You now have the file on disk. A typical file with 64-byte records would appear as shown in Figure RDFB-1. Notice that there are "gaps" in the records because some of the records were not written out by a PUT. These gaps can be filled in by subsequent PUTs with new data or existing records can be overwritten.

Figure RDFB-1 — *File with 64-byte Records*



General Flow for Reading in a Random File: To read in an existing random file, do this:

1. OPEN the file as in 1 above, for example

```
100 OPEN "R",2,"INVENTOR/APR",64
```

2. Read in a record by a GET statement. This statement is the opposite of PUT and reads in a specified record number into the disk buffer.

```
110 GET 2,I
```

for example, reads in the Ith record into disk buffer 2 associated with the file "INVENTOR/APR". I, of course, could be 23, 56, 1, 34, or any record number in the file, as long as it exists.

3. Read the fields in the record as follows: You can simply set a string variable to the field name. The string variable might have leading or trailing blanks, depending upon whether you had specified LSET or RSET to initially store the variable.

```
120 CH$=OH$          'read on hand # from record
```

Convert the ASCII data in the record to numeric by the funtions CVI, CVS, and CVD. These functions are the opposite to MKI$, MKS$, and MKD$ — they convert the character data to integer, single-precision or double-precision variables. To convert the remaining fields to numeric, you'd have:

```
130 CP=CVS(PN$)      'get part number
140 CS=CVS(SC$)      'get supplier code
150 CH=CVS(OH$)      'get # on hand
160 CO=CVS(OO$)      'get # on order
```

4. Repeat the GET and processing for as many records as required. In some cases, you'd know how many records need to be GETted (terrible! GOTten?). In other cases, you'd be going through all of the file. You can get the "end-of-file" number of the last record in the file by:

```
170 I=LOF(1)
```

which sets I to the number of the last record for the file associated with disk buffer 1.

5. CLOSE the file as in the Write case.

To Append Data to an Existing File: To add records to an existing file, simply do a PUT with a new record number. If the record number is beyond LOF, the last record number, more disk space will be allocated, based on the record size. Suppose that you had 10 128-byte records out on disk as file "RATTAIL". You could do a PUT to record 20:

```
200 PUT 1,20
```

and "RATTAIL" would be increased in size on disk to 2560 bytes minimum. The actual size would depend upon how many granules (see DSHL) were involved to hold the 2560 bytes. LOF would also be adjusted.

RDHT

Random Data, How to Generate

Random data is used primarily for simulations. What's a simulation? Suppose that you're writing a game called "Fort Worth." It's about a dynamic electronics company with a full complement of charismatic engineers, programmers, and (especially) marketing people. You'd like to simulate random occurences of Texas tornados to make a challenging and interesting game. How do you do it?

The RND function allows you to generate a type of random number between 0 and 32767. Using

```
100 A=RND(1)
```

will produce values of .65478.., .53421.. and .76512.. – the numbers will be between 0 and 1.

If you use an "argument" other than 1, numbers between 1 and that argument will be generated. Using

```
100 A=RND(1000)
```

for example, will generate integer numbers like 34, 678, 1000, 4, 45 and so forth.

You can use these random numbers to simulate events. Back to our "Ft. Worth" game, for example . . . You might want to throw in a tornado every 10 passes through the game or so to make it interesting by using something like

```
100 IF RND(10)=5 THEN PRINT "TORNADO"
```

In the above example, RND(10) would be equal to 5 about 1 out of 10 times, and you'd be simulating the random occurence of a tornado.

The numbers generated by RND are not truly random. Over thousands of passes, you'd tend to get a fairly equal distribution of all numbers – an equal number of 1s, 2s, 3s and so forth, up to the largest number specified. However, the number sequence is *repeatable.* Eventually the sequence would repeat, although the "cycle" would be tens of thousands of numbers long. Once it did repeat, though, the exact same sequence of numbers would be printed. Also, if you had a way to start from a given number, RND would generate the same sequence. This type of random number generation is called "pseudo-random," as the number sequence repeats the same sequence from a given value.

The starting number of the sequence is known as the "seed" value. If you power up your system and then PRINT RND(1000), you'll get the same number each time. The "seed" value that the system starts with is constant.

In the Model I/II/III (except for I/III, Level I), you can "*reseed*" the random number generator by using the RANDOM command. RANDOM sets the starting seed to a true random number by using a random number from the I register, which is constantly cycling from 0 through 255. The RANDOM command doesn't exist in the Color Computer, but use RND(TIMER) to seed the random number generator initially to a random value. (TIMER values are derived from the real-time clock in Extended and Disk BASIC).

Models I/III: Now, suppose that you've got the opposite problem. What if you want to repeat the same sequence of pseudo-random numbers? How do you start from the same seed value? POKE the seed value into locations 16554, 16555 and 16556. (Remember that these may change in subsequent versions of ROM . . .)

For example,

```
100 POKE 16554,0:POKE 16555,0:POKE 16556,0
110 PRINT RND(1000),
120 GOTO 110
```

always generates the same sequence of 3, 47, 27 . . .

RDLD
Routing Devices, Model I/III LDOS

The ROUTE command is used to route logical devices (see LDIS). ROUTE will reroute all input/output that would be acquired from one logical device and inform the system that it is to be acquired from another logical device. Similarly, it will change a destination device from one device to another. Finally, it will substitute a disk file name for a logical device.

```
ROUTE *PR TO FILEB:0
```

for example, routes all I/O that would normally be sent to the system print device *PR to a disk file called FILEB:0. The ROUTE command "opens" the file. Subsequent printing in a BASIC program, assembly, or other LDOS-compatible program will go to the disk file. When you determine that the routing is complete, the disk file must be closed by an LDOS

```
RESET *PR
```

command.

Similarly, you can also route the *CL (comm line) input to take the place of the *KI (keyboard input) device:

```
ROUTE *CL TO *KI
```

or route a disk file contents to the *CL output:

```
ROUTE SANDWICH/APR TO *CL
```

Always RESET the logical device after routing is over.

ROUTE is extremely handy for moving the flow of data around, especially to and from disk files. Up to 4 system devices can be routed simultaneously.

RGER
RG Error

Return Without GOSUB error.

Generally, every time you execute a GOSUB in BASIC, you must have a RETURN somewhere later in the program. You must never have executed more RETURNs than you have executed GOSUBs. If you have "nested" GOSUBs, check your code to see that there is a RETURN for every GOSUB.

RHS1
RS-232-C Interface, How to Set, Model I

Read RSWI if you are unfamiliar with data communications.

The RS-232-C interface board contains 9 separate switches that must be set properly to communicate with an RS-232-C device. The switches are shown in Figure RHS1-1.

Follow these steps to set up the interface.

1. In general, Model I software drivers for modems, line printers, and other devices, *do* read the RS-232-C sense switches, but it depends upon the software. The sense switches shown in the figure may be ignored and the parameters defined by the switches may be set under program control. Read the documentation for the RS-232-C program to see if the switches need to be set. In any event, you must set the COMM/TERM switch properly and cannot lose anything by setting the sense switches to "nominal" settings.

LDOS users: The SET *CL TO RS232R/DVR command enables you to set the switches in software when using the LDOS LCOMM program or other communications programs that run under LDOS. You must set COMM/TERM, however, as follows.

2. The COMM/TERM switch.

A. The only purpose of this switch is to swap the RD and TD lines. The RD and TD lines (see RSWI) stand for "read data" and "transmit data" and are control data being received by the Model I (RD) and data being transmitted by the Model I (TD). Setting the COMM/TERM switch to TERM sets up the Model I as a "terminal" and is the normal setting for using the Model I to communicate with Bulletin Board systems (see BBUS) CompuServe or the Source (see CPSU), or as an answering device for other "networking" situations. Setting the COMM/TERM switch to COMM sets up the Model I as a transmitting device and is normally used when the RS-232-C port is used to connect a serial printer, plotter, or other subsidiary input/output device. Confusing? Yes. For the record: RD and TD are truly RD and TD when COMM/TERM is set to TERM, and swapped when COMM/TERM is set to COMM.

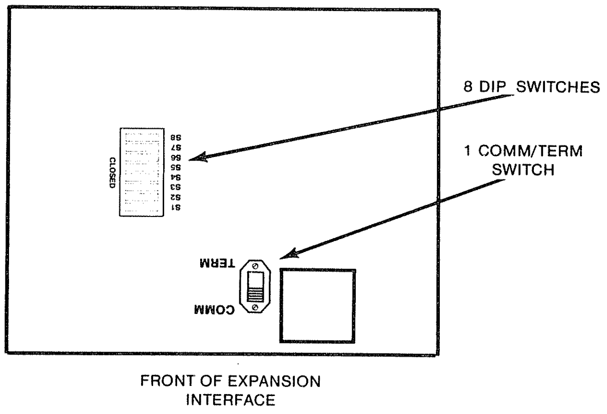
R

B. Remove the RS-232-C cover by unscrewing the four screws that hold the cover in place. If you will be doing extensive RS-232-C work, consider leaving the screws off; the monitor will fit nicely on the top of the Model I expansion interface and the screws are not required.

C. You are now looking at the dreaded and misunderstood RS-232-C board. Set the COMM/TERM switch towards the back for COMM and towards the front for TERM. Leave the cover off and go to step 3.

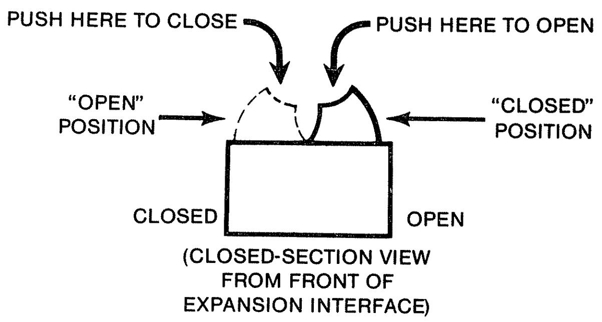
3. The RS-232-C Sense Switches.

Figure RHS1-1 – *Model I RS-232-C Board Switches*



A. These are 8 "dip switches" that can be set by a point of a sharp object, such as a small Phillips-head screwdriver. Listen for a firm click and make certain the dip switch is all the way down to either the CLOSED or OPEN position, as shown in Figure RHS1-2.

Figure RHS1-2 – *Dip Switch Setting*



B. If you will look closely at the board, you will see "CLOSED" or "OPEN" on the printed circuit "etch", or use Figure RHS1-1.

C. There are 5 RS-232-C parameters selected by the sense switches (see RSWI), baud rate, parity enable, number of stop bits, word length and parity select. Before we go on, let me give you some guesses as to proper switch settings:

Bulletin Board Systems, CompuServe, Source, Etc.:300 baud, parity disabled, 1 stop bit, 8-bit word length, odd parity. (TERM/COMM to TERM.)

NEC Printers, other "serial" printers: 300 baud, parity disabled, 2 stop bits, 7-bit word length, odd parity. (TERM/COMM to COMM.)

Try these if you must, but I'd advise reading on.

D. Baud rate. Set by the 3 switches shown in Figure RHS1-3. Use these settings

Baud Rate	S6	S7	S8
11Ø	C	C	C
15Ø	C	C	O
3ØØ*	O	C	C
6ØØ	O	C	O
12ØØ	C	O	C
24ØØ	C	O	O
48ØØ	O	O	C
96ØØ	O	O	O

*=typical

E. Parity enable. Set the one switch shown in Figure RHS1-4. Set closed for parity enabled, open for parity disabled. For many applications "parity" (see RSWI) will be disabled and the "parity select" will be meaningless.

F. Stop bits. Set the one switch shown in Figure RHS1-5. One stop bit is closed, two stop bits is open. In general, one stop bit is typical. See RSWI.

G. Word length. Set the two switches shown in Figure RHS1-6.

In general, devices requiring ASCII data only use word lengths of 7 bits, while other devices use 8 bits. Five and 6 bits seldom used except in archaic teletypewriters. Try 8 bits if unknown.

Figure RHS1-3 – *Baud Rate Settings*

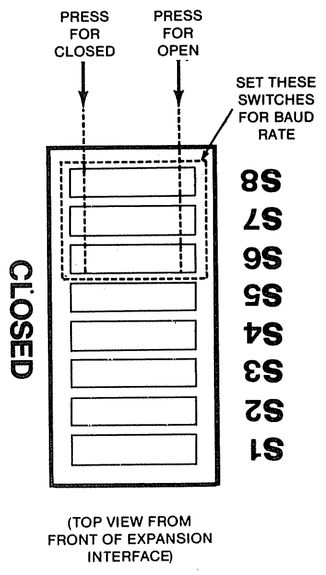

PRESS FOR CLOSED
PRESS FOR OPEN
SET THESE SWITCHES FOR BAUD RATE
CLOSED
S8
S7
S6
S5
S4
S3
S2
S1
(TOP VIEW FROM FRONT OF EXPANSION INTERFACE)

Figure RHS1-4 – *Parity Enable Setting*

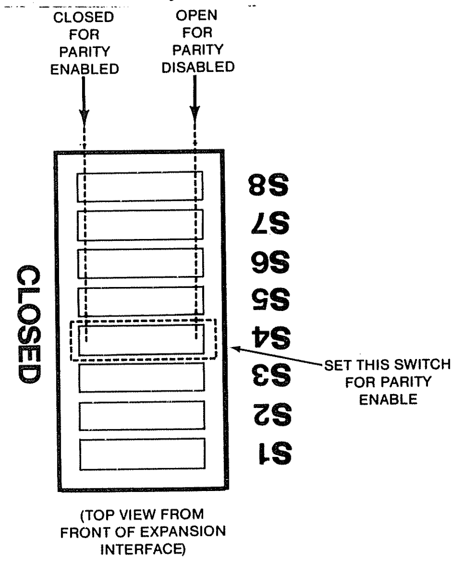

CLOSED FOR PARITY ENABLED
OPEN FOR PARITY DISABLED
SET THIS SWITCH FOR PARITY ENABLE
CLOSED
S8
S7
S6
S5
S4
S3
S2
S1
(TOP VIEW FROM FRONT OF EXPANSION INTERFACE)

Figure RHS1-5 – *Stop Bit Setting*

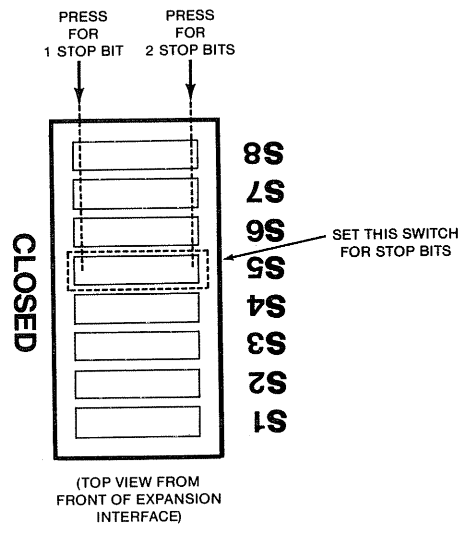

PRESS FOR 1 STOP BIT
PRESS FOR 2 STOP BITS
SET THIS SWITCH FOR STOP BITS
CLOSED
S8
S7
S6
S5
S4
S3
S2
S1
(TOP VIEW FROM FRONT OF EXPANSION INTERFACE)

Figure RHS1-6 – *Word Length Setting*

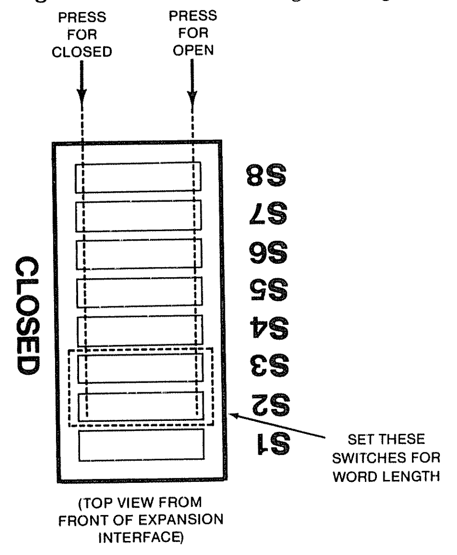
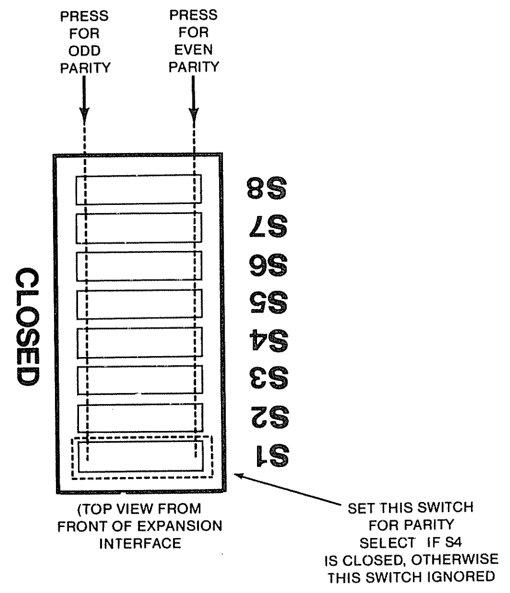

PRESS FOR CLOSED
PRESS FOR OPEN
SET THESE SWITCHES FOR WORD LENGTH
CLOSED
S8
S7
S6
S5
S4
S3
S2
S1
(TOP VIEW FROM FRONT OF EXPANSION INTERFACE)

Figure RHS1-7 – *Parity Odd/Even*


PRESS FOR ODD PARITY
PRESS FOR EVEN PARITY
SET THIS SWITCH FOR PARITY SELECT IF S4 IS CLOSED, OTHERWISE THIS SWITCH IGNORED
CLOSED
S8
S7
S6
S5
S4
S3
S2
S1
(TOP VIEW FROM FRONT OF EXPANSION INTERFACE

Word Length	S2	S3
5 bits	C	C
6	C	O
7	O	C
8	O	O

H. Glad to see you've made it thus far. Have courage.... Parity select. If you've disabled parity above, set to closed or open; it makes no difference. If you have enabled parity, read RSWI and set to open (even) or closed (odd). Typical setting is either, with parity disabled. See Figure RHS1-7.

I. Double check those switch settings!

4. You can now note your switch settings on a piece of paper, put the RS-232-C cover back on without the screws, and continue with procedures on modems, acoustic couplers, or other "Serial" devices. Special note: In lieu of changing the TERM/COMM switch when you connect to a serial printer and then to a modem, consider making up a special cable, one for each of your serial devices.

RHS3
RS-232-C Interface, How to Set, Model III

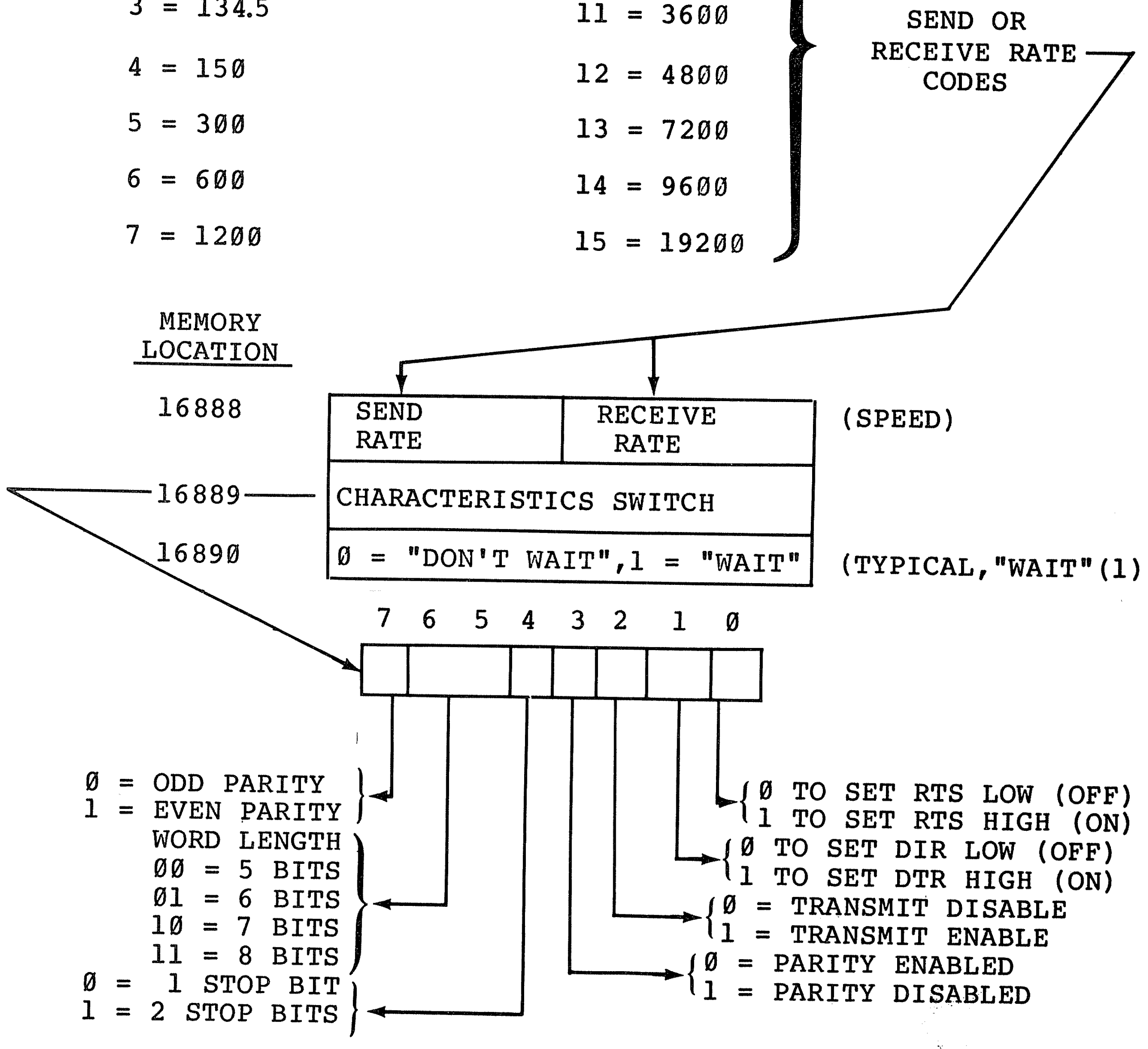
Read RSWI if you are unfamiliar with data communications.

The RS-232-C interface is set completely under program control; there are no switches to set manually, as in the Model I. If you are using a precanned program, such as VideoText from CompuServe, then the program will setup the RS-232-C for you; refer to the documentation in the program for connection information. If you are not using a precanned program and want to do your own serial interfacing, then continue.

Model III, no disk: The RS-232-C interface is set by changing three locations in RAM, either by BASIC POKEs (see PPKU) or by assembly language code. These three locations are shown in Figure RHS3-1, along with their contents.

Figure RHS3-1 – *RS-232-C Locations*

CODE	BAUD RATE	CODE	BAUD RATE
Ø =	5Ø	8 =	18ØØ
1 =	75	9 =	2ØØØ
2 =	11Ø	1Ø =	24ØØ
3 =	134.5	11 =	36ØØ
4 =	15Ø	12 =	48ØØ
5 =	3ØØ	13 =	72ØØ
6 =	6ØØ	14 =	96ØØ
7 =	12ØØ	15 =	192ØØ



To send a character in BASIC, perform the following steps:

1. Setup locations 16888, 16889, and 16890 to define the RS-232-C parameters.

2. POKE 16526,85 : POKE 16527,0 to setup a USR call to the ROM driver for "send an RS-232-C character" at location 85 (**55H**).

3. Store the character to be sent in location 16880 by a POKE.

4. Make this USR call to send the character:

```
100 A=USR(0)
```

5. Repeat steps 3 through 4 for other characters as required.

To receive a character in BASIC, perform the following steps:

1. Setup locations 16888, 16889, and 16890 to define the RS-232-C parameters. *Use the "wait" option.*

2. POKE 16526,80 : POKE 16527,0 to setup a USR call to the ROM driver for "receive an RS-232-C character" at location 80 (**50H**).

3. Make this USR call to receive the character:

```
100 A=USR(0)
```

4. When the return is made from the call, the received character will be in location 16872. Do

```
200 A$=CHR$(PEEK(16872))
```

to get the character.

5. Repeat steps 3 and 4 for additional characters.

Model III, TRSDOS: Use the TRSDOS SETCOM command to define the RS-232-C parameters. The format is

```
SETCOM (WORD=w,BAUD=b,STOP=s,PARITY=p,MODE=WAIT)
```

The w parameter is 5, 6, 7, or 8. The baud rate is defined by b and should be a standard baud rate as shown in Figure RHS3-1. The number of stop bits, s, should be 1 or 2. The parity, p, is 1 (odd), 2 (even), or 3 (no parity).

If the mode is WAIT, any call to read an RS-232-C character will wait until the character has been received before returning. If NOWAIT is specified, a return will be made without a character. The NOWAIT option is used for "interleaving" read characters with transmission of characters.

SETCOM (OFF) turns off the RS-232-C.

A typical SETCOM for CompuServe (see CPSU) might be

```
SETCOM (WORD=8,BAUD=300,STOP=1,PARITY=3,MODE=WAIT)
```

A SETCOM without parameters displays the current settings.

The ROM calls to "Read a Character (**50H**)" and "Send a Character (**55H**)" work as described under "Model III, no disk".

Model III, LDOS: Use the SET command as follows:

```
SET *CL TO RS232T/DVR (BAUD=b,WORD=w,STOP=s,PARITY=p)
```

The Baud rate, b, is as shown in Figure RHS3-1. The w parameter is 5, 6, 7, or 8. The s parameter is 1 or 2 stop bits. Parity, p, is ON or OFF. If ON, you may specify ODD or EVEN (. . . PARITY=ON,EVEN).

Typical settings for CompuServe (see CPSU) might be

```
SET *CL TO RS232T/DVR (BAUD=300,WORD=8,STOP=1,
PARITY=OFF)
```

LDOS also provides other parameter options — BREAK to recognize a break condition, the ability to turn DTR and RTS on or off, and the ability to look for DSR, CD, RTS and RI on or off. These options are very useful for matching "line conditions" for "foreign" (non-Radio Shack) devices.

RHTU
Resistors, How to Use

Ah, these electronic parts for computer circuits . . . A resistor "resists" the flow of current (A lot of these terms date from Faraday) much in the same way that reducing the size of a water pipe restricts the flow of current.

Resistors generally required in computer circuits carry little current, and can therefore be physically small. Popular sizes are 1/4 watt and 1/8th watt. You can always use a *larger* wattage rating, but should never use a smaller.

Besides the power (watts) rating, resistors are classified by resistance value and precision. The resistance value is measured in *ohms*. One ohm restricts the flow to one ampere with a voltage of 1 volt:

I (current in amperes) = E (voltage in volts) / R (resistance in ohms)

Typical resistances in computer circuits are 100 ohms through 1,000,000 ohms (1 megohm).

The precision of a resistor is called "tolerance." Typical resistors are 10% or 5% tolerance. Precision resistors are 1% tolerance. The percentage indicates how much the resistance can vary — a 5%, 100-ohm resistor can vary between 95 and 105 ohms, for example. Usually 10% tolerance resistors are fine.

Resistors are made out of various materials. Typical computer circuit resistors are "carbon-film." If you need resistors, go to Radio Shack and use 5%, carbon-film, ¼-watt resistors; they'll suffice for just about every application in your system — there's not a great deal that you can do wrong with resistors. (Where's that smoke coming from?) . . .

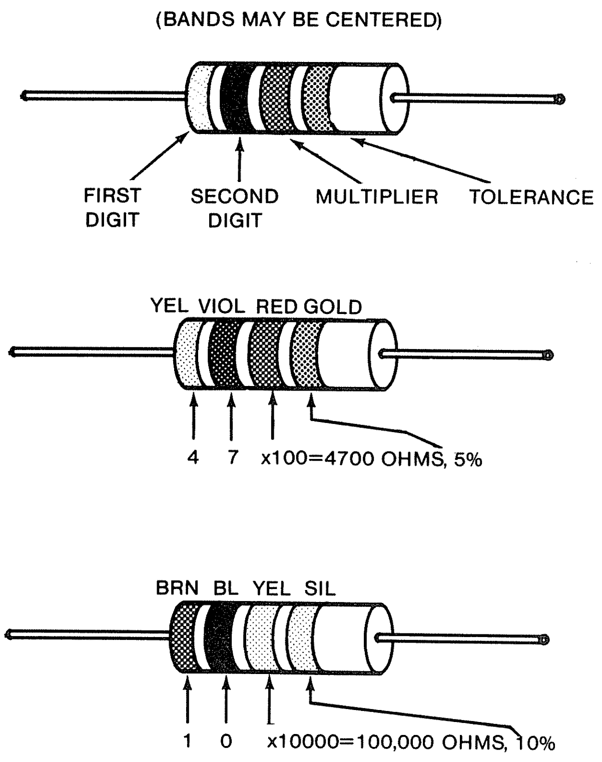
Resistors are not usually marked with resistance values. A color code is used, as follows:

Black	Ø or X1
Brown	1 or X1Ø
Red	2 or X1ØØ
Orange	3 or X1ØØØ
Yellow	4 or X1ØØØØ
Green	5. or X1ØØØØØ
Blue	6 or X1ØØØØØØ
Violet	7 or X1ØØØØØØØ
Gray	8 or X1ØØØØØØØØ
White	9 or X1ØØØØØØØØØ
Gold	5% tolerance
Silver	1Ø% tolerance
None	2Ø% tolerance

The tolerance "band" is the last band, as shown in Figure RHTU-1. A yellow, violet, red, silver banded resistor would be 470 ohms, 10%.

The color code has a mnemonic (established by Thomas Edison, no doubt) that has been cleaned up here: Bad Boys Roust Our Young Girls But Violet Grins Wildly.

Figure RHTU-1 – *Resistor Color Code Banding*



RLIB
Replacing a Line in BASIC

The simplest way to replace a BASIC program line: While in BASIC, reenter the line with the same line number and modified commands. BASIC will replace the old BASIC line with the new BASIC line.

Another way to modify a longer line: Use the Edit mode in BASIC (see procedure EMBH).

RNBL
Renumbering BASIC Lines, Non-LDOS Systems

BASIC lines can be renumbered on all systems so that additional lines can be inserted, or so that gaps in the line numbering can be deleted to make the listings look "pretty." On all renumbering schemes, all references to the line numbers on GOSUBs and GOTOs will be changed within the program. The renumbered program should run exactly like the old.

Model I, LDOS: See RNBM.

Model I, RS: Sorry people, there is no renumber command included in either Level II or Disk BASIC.

Model II: Use the RENUM command in BASIC. The format of RENUM is:

```
RENUM newline,startline,increment
```

The new line number is the first line number of the renumbered program. The start line number is the line number in the original program where renumbering is to start. The increment is the line number increment to be used. All parameters are optional. The "default" parameters are 10 for new line, the first program line of the original program for the start line number, and an increment of 10.

Model III, no disk: No renumber capability.

Model III, TRSDOS Disk BASIC: Use the NAME command in BASIC. The format of NAME is

```
NAME newline,startline,increment
```

The new line number is the first line number of the renumbered program. The start line number is the line number in the original program where renumbering is to start. The increment is the line number increment to be used. All parameters are optional. The "default" parameters are 10 for new line, the first program line of the original program for the start line number, and an increment of 10.

Model III, LDOS: See RNBM.

Color Computer, Color BASIC: No renumber capability.

Color Computer, Extended BASIC and Disk BASIC: Use the RENUM command in BASIC. The format of RENUM is:

```
RENUM newline,startline,increment
```

The new line number is the first line number of the renumbered program. The start line number is the line number in the original program where renumbering is to start. The increment is the line number increment to be used. All parameters are optional. The "default" parameters are 10 for new line, the first program line of the original program for the start line number, and an increment of 10.

Example: For all Radio Shack renumbers,

```
RENUM
```

renumbers the entire BASIC program. The first new line is 10, and the increment is 10.

```
RENUM 1ØØ
```

renumbers the entire BASIC program. The first new line is 100, and the increment is 10.

```
RENUM 1ØØ,2ØØ
```

renumbers the old program lines from 200 on. Line 200 will become 100, the next line after 200 will become 210, etc.

```
RENUM 1ØØ,,2Ø
```

renumbers the entire BASIC program. The first new line is 100, and the increment is 20 (note use of commas to specify the "default").

RNBM
Renumbering BASIC Lines, Model I/III LDOS

To renumber an entire program with the new line numbers starting from 20 with an increment of 20, do the following:

```
CMD"N"
```

This command will renumber only if there are no errors (such as undefined line numbers). If there are errors, no lines will be changed.

If you're certain there are no errors, you can use the

```
CMD"N !"
```

form of renumber. This will skip the check for errors. My recommendation: don't skip the check.

To renumber the entire program with your own new line number and increment:

```
CMD"N ! 1,newline,increment,6552 9"
```

Put your own starting line number in newline and your own increment in increment.

To renumber a program block of lines:

```
CMD"N ! startline,newline,increment,endline"
```

This command will renumber from a startline in the original program through an endline in the original program. The new line numbers will start with newline and will have an increment of "increment." Confused? Suppose you had a program of lines 100 through 1000 with increments of 10. If you wanted to change line numbers 500 through 600 to 550 through 570 with increments of 1, you'd have:

```
CMD"N ! 5ØØ,55Ø,1,6ØØ"
```

You can see that this LDOS command, like all LDOS commands, gives you plenty of options. You can renumber any block of lines within a BASIC program. However, if the renumbering would result in the lines being out of sequence, you'll get a BAD PARAMETERS ERROR. An example is trying to renumber those lines at 500 through 600 with a newline of 50; a sequence error would result.

ROOF
Rounding Off in BASIC, Most Systems

BASIC is a little bit too exact. Who wants a checking account balance of $123.7672! Because BASIC will attempt to be as precise as possible you'll have to handle such things as "formatting" the results to something more reasonable ($123.77 in the above example) and "rounding off" cents.

First method: You can use the *PRINT USING* (see PRUU) and *LPRINT USING* (PRUU) to "truncate," or lop off extra digits in cents or other numeric values.

Second method: Test the fractional part of the numeric value yourself and round up or down. The scheme goes something like this for dollars and cents (A is the numeric dollars and cents value, such as 21.7325):

```
1ØØ X=A+.ØØ5
11Ø A$=STR$(A)
12Ø FOR I=1 TO LEN(A$)
13Ø IF MID$(A$,I,1)="." THEN GOTO 16Ø
14Ø NEXT I
15Ø A$=A$+".ØØ"
16Ø IF I=LEN(A$) THEN A$=A$+"ØØ" ELSE IF I=LEN(A$)-1
    THEN A$=A$+"Ø" ELSE IF I<>LEN(A$)-2
    THEN A$=LEFT$(A$,I+2)
```

This will always give you a rounded off dollars and cents value in A$ at the end; there is a better way, however.

Third method: This is the better way:

```
1ØØ A$=STR$(A)
11Ø PRINT USING "$$###,###.##";A$
```

You'll have to refer to PRINT USING (PRUU) to hone this one to a sharp edge; as it stands, it will give you a display of a "floating dollar sign" and up to $999,999.99, rounded off. The PRINT USING decimal point automatically invokes a round off.

You can throw all those fractional cents into my account, by the way. If enough of you do it, I won't have to depend upon sales for this book ...

RSLD
RESETting the System, Model I/III LDOS

Use RESET to reset any LINKing (see LDLD), ROUTEing (see RDLD), SETting, or filtering of system devices. A

```
RESET
```

is a "global" reset that resets the world. The RESET also resets memory usage. Certain LDOS functions, such as SPOOL (see SPHT), use high memory, and a global RESET will release this memory area.

To reset only a single device use a RESET with a logical device name (see LDIS) as in

```
RESET *PR
```

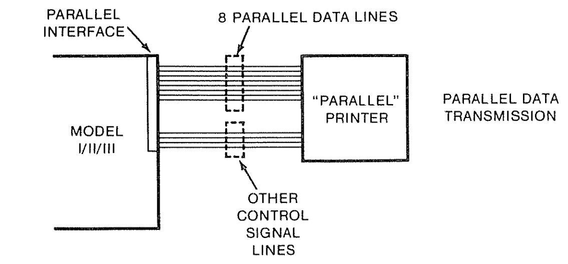
RSWI
RS-232-C, What Is It?

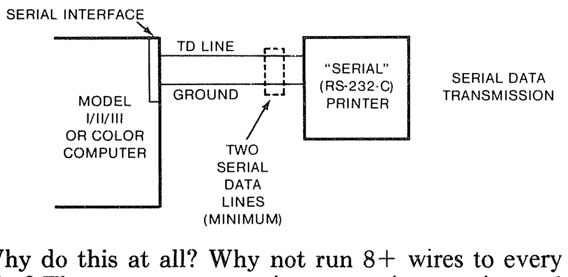
Before you can expect to connect a serial printer or modem to your RS-232-C interface, you've got to know some of the rules. We'll keep it simple (you've heard of "KISS," haven't you — "Keep It Simple, Stupid!" — "stupid" being the writer who attempts to impress rather than communicate) . . .

The RS-232-C interface on the Models I, II, III, and Color Computer is a standard interface for connecting "serial" devices. ("RS-232-C" is sometimes called "RS-232" or "serial," "serial port" or "asynchronous transmission"). There is one RS-232-C connector on any of the four models except for the Model II, which has two RS-232-C connectors. In the case of the Model I/III and Color Computer, therefore, you can connect only one serial device at a time. The connector is a special 25-pin connector that is readily available. Cables to connect standard Radio Shack devices and some common devices are also available from Radio Shack or other electronic stores. In some cases, if you have purchased a "foreign" (non-Radio Shack) device, you may have to wire up (or have wired) a special cable.

The RS-232-C interface transmits data to a serial device such as a printer by converting an ASCII (see ADFW) byte or other byte of data into a series of bits. Instead of needing 8 wires for the 8 bits making up a byte (plus some other wires for "ground" and controlling signals), the data can be transmitted with only two wires, as shown in Figure RSWI-1.

Figure RSWI-1 – *Two-Wire Data Transmission*



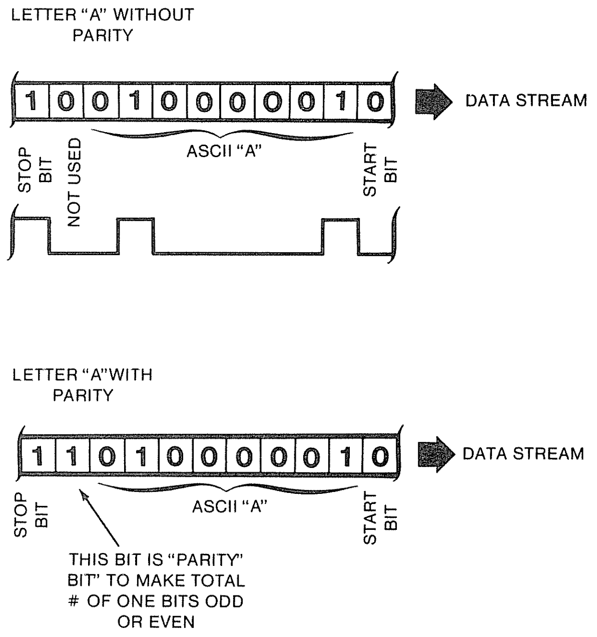

SERIAL INTERFACE
TD LINE
GROUND
MODEL I/II/III OR COLOR COMPUTER
"SERIAL" (RS-232-C) PRINTER
SERIAL DATA TRANSMISSION
TWO SERIAL DATA LINES (MINIMUM)

Why do this at all? Why not run 8+ wires to every device? Three reasons: two wires are easier to string and cost less. The signals are more immune to electrical noise and can be strung longer distances. The signals can be converted into audio tones and transmitted via telephone lines very easily (see MWAT). There are other reasons, but these are some of the most important.

Now look at Figure RSWI-2. You'll notice in the figure that the 8 data bits are in the center of the group of bits transmitted in "serial." The 8 data bits are "bracketed" by a leading "start" bit and one or two "stop" bits. The start bit always appears for any RS-232-C transmission. Similarly, there are always one or two stop bits. This is a good time to add that format of the RS-232-C transmission — number of stop bits, total number of data bits, and so forth — is usually predefined before any transmission starts and changes only for different types of serial applications.

The last bit before the stop bit(s) may be an optional with farm prices) . . . . The parity bit is a kind of "checksum" bit that can be tested to see if any data bit has been erroneouslly sent. If the parity bit is used, and "even" parity has been set, then the parity bit is set to make the total number of 1 bits in the data bits an even value, as shown in Figure RSWI-2. If the parity bit is used, and "odd" parity has been set, then the parity bit is set to make the total number of 1 bits in the data bits an odd value, as shown in the figure. Should parity be used? It depends upon the data. If you are reading messages from a Bulletin Board system (see BBUS) an occasional parity error isn't that important. If you are "downloading" programs from a large system into your TRS-80, then you'll want to use parity. Many times you won't be given the choice because the format has been established by the network or input/output device.

Figure RSWI-2 – *Parity Bit in Data Transmission*



The number of data bits in between the start bit and parity (or stop bit(s)) may be 5, 6, 7 or 8. Usually 7 (for ASCII) or 8 bits is used. A typical configuration is 1 start bit (always a binary 0), 8 data bits, no parity bit and 1 stop bit (always a binary 1), making a total of 10 bits per character with 8 of those being the actual data bits.

The spacing between each of the bits in a byte or character is relatively precise. The transmitting and receiving RS-232-C device each have an internal "clock" which controls this "bit time" spacing. The "bit times" determine the rate at which data can be sent and establish the "baud rate." There are a number of "standard" baud rates. Standard is in quotes because while these rates are most common, other baud rates can be used. The standard baud rates, bit times, format and data characters per second are:

Baud Rate	Bit Time	Format	Data Chars/Sec
11Ø	9.Ø9 ms	SDDDDDDDXSS	1Ø chars/sec
15Ø	6.Ø6 ms	SDDDDDDDXS	15 chars/sec
3ØØ	3.33 ms	SDDDDDDDXS	3Ø chars/sec
6ØØ	1.66 ms	SDDDDDDDXS	6Ø chars/sec
12ØØ	.833 ms	SDDDDDDDXS	12Ø chars/sec
24ØØ	.416 ms	SDDDDDDDXS	24Ø chars/sec
48ØØ	.2Ø8 ms	SDDDDDDDXS	48Ø chars/sec
96ØØ	.1Ø4 ms	SDDDDDDDXS	96Ø chars/sec

S=Start bit D=Data bit X=Parity or Data Bit
S=Stop Bit ms=milliseconds or 1/1ØØØths second

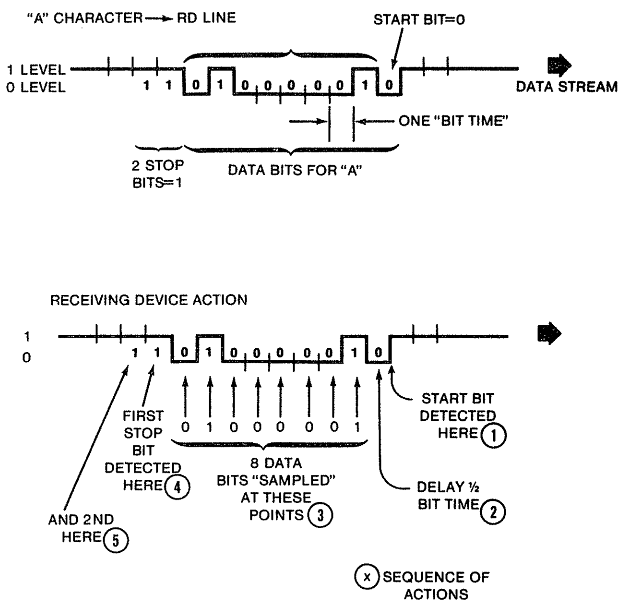
You can see from the table that the baud rate, bit time, number of bits per transmission,and number of characters per second are all interrelated. For a 10-bit format (SDDDDDDDXS), the number of characters per second is the baud rate divided by 10.

The bit times are rigidly fixed at both the receiving and transmitting ends. The receiving RS-232-C device looks for the "start" bit on the "RD" or "receive data" line. This line is normally held at a logic 1. When the start bit 0 comes through, the receiving device waits one/half bit time, and then reads in the 10 or 11 bits of the character or data byte. (The receiving and transmitting device know the baud rate and format by prearrangement). The last bit is a stop bit and resets the RD line to a logic 1. This scheme is shown in Figure RSWI-3

Earlier we called RS-232-C "asynchronous" transmission. Asynchronous means at unpredictable times. Although the receiving device "syncs up" to receive the bits from the transmitting device and expects them at precise bit times, it does not know when the next character or byte will be transmitted. It only knows when it receives the next start bit and starts the whole process over again. If a hunt and peck typist (such as myself) is sending data from a keyboard, each burst of 10 bits will occur about every 1/3rd of a second as shown in Figure RSWI-4, but I may pause for 10 seconds to drink Bushmill's Irish Whiskey (writers have an image to preserve). Since characters occur at unpredictable times, this type of transmission is "asynchronous" between characters or bytes.

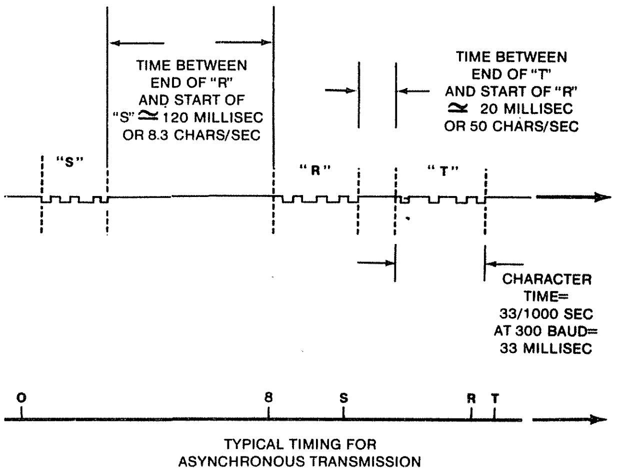
We're almost done... The receiving device converts the stream of serial bits to a byte of 8 data bits, which is transmitted in parallel to the CPU in the TRS-80. The transmitting device is a parallel-to-serial converter, while the receiving device is a serial-to-parallel converter.

Figure RSWI-3 – *Asynchronous Data Transmission*



If we take the two wires of Figure RSWI-1 and add a third, called RD (Receive Data), we can send data either direction. A printer may require only two wires (ground and TD), but a modem requires at least three wires (ground, TD, and RD) as data must be sent in both directions. As a matter of fact, in the Model I, II, III and Color Computer, data can be sent in both directions *at the same time.* This operation is called "full-duplex".

Figure RSWI-4 – *Character Transmission*


TIME BETWEEN END OF "R" AND START OF "S" ≃ 120 MILLISEC OR 8.3 CHARS/SEC
TIME BETWEEN END OF "T" AND START OF "R" ≃ 20 MILLISEC OR 50 CHARS/SEC
"S"
"R"
"T"
CHARACTER TIME= 33/1000 SEC AT 300 BAUD= 33 MILLISEC
0
8
S
R T
TYPICAL TIMING FOR ASYNCHRONOUS TRANSMISSION

In 'half-duplex' mode, data can only be sent in one direction. Many times modems are run in full duplex to 'echo back' the typed character. It appears that the character being typed is also being displayed, but in fact, the receiving device is sending back the character and the software is receiving the character and displaying it. Why do this? Because errors in transmission will be apparent. With echo back you'll know for certain the receiving device got the character properly.

One more brief description, and I'll let you go. This is for hardware types only, however. Others can skip this part. The common RS-232-C signal conventions are logic 0 greater than +3 volts and logic 1 less than -3 volts. On the Models I, II, III, and Color Computer, +12 and -12 volts are used for logic 0 and 1, respectively. Note that these voltages are inverted from other logic where a positive voltage is a 1 and a negative voltage is a 0.

Refer ro RHS1, RHS3, RSWI, and others for further related topics on RS-232-C interfacing.

RTCN
Real Time Clock No Longer Accurate — Why?

TRSDOS and LDOS on the Models I and III both turn off the real-time-clock when they are doing cassette or disk input/output. These I/O operations are done in a timed program loop. During this timing, no interrupts can occur to add unexpected "overhead" to the timing, hence the rtc is turned off. LDOS, being a more fastidious operating system, informs you via the message above. Time lost may be on the order of dozens of seconds, depending upon the operations. There is no solution to this problem other than resetting the real-time-clock by the TIME command.

RTCT
Real-Time Clock, To Turn On, Model I/III Disk BASIC

To turn on the real-time clock display, type

```
CMD"R"
```

The current time will be displayed in the upper right-hand corner of the screen, and will update every second.

Type

```
CMD"T"
```

to turn off the display from Disk BASIC.

Model III: Note that the real-time-clock is always running and that these commands simply enable or disable the display.

See TSRT to set the time.

RTDB
Return to TRSDOS/LDOS from BASIC, Model I/III

Type

```
CMD"S"
```

while in BASIC command mode.

RUOF
ROUTE, Use of, Model III TRSDOS

Oh, oh! You missed the little errata sheet attached to your Model III TRSDOS manual.

ROUTE is not implemented in Model III TRSDOS 1.3.

RUTP
RUNning a BASIC Program, All Systems

Enter RUN, and your BASIC program will start from the beginning. Enter RUN nnnn, and the program will run from line nnnn.

RVCC
Reverse Video, Color Computer

If green on black characters appear in the middle of entering data from the keyboard, you've inadvertantly pressed the SHIFT and "0" keys simultaneously. Pressing these two keys again will restore normal black on green.

S1EA
Using the Series I Editor Assembler, Model I/III

The Series I Editor Assembler is a Z-80 Editor/Assembler package available on either cassette or disk. No DEBUG capability is provided. You can edit and assemble Z-80 assembly language programs with the Series I (see ALWI for a description of assembly language).

To Load the cassette version, Model I/III, Level I:

1. Turn on your system and get READY.

2. Load the SYSTEM cassette into the recorder and type CLOAD followed by ENTER. The cassette should load.

3. After about 2 minutes, you should see the prompt "PRESS ENTER WHEN CASSETTE IS READY."

4. Load the EDTASM cassette into the recorder and press ENTER. The cassette should load.

5. After about 2 minutes, you should see an asterisk (*) prompt. EDTASM is now loaded.

To load the cassette version, Model I/III, Level II/III:

1. Load the EDTASM cassette as a SYSTEM tape with the name "EDTASM" by following the procedure in LMFN.

2. An asterisk (*) indicates a successful load.

To load the Disk version, Model I/III:

1. After

```
TRSDOS READY
```

type EDTASM, ENTER.

2. An asterisk (*) indicates a successful load.

General Description:

The EDTASM package consists of two parts, the Editor and Assembler.

The Editor is used to construct or modify "source" files that are largely ASCII files (see AFWA) with the exception of some non-standard characters for line numbers. The source files are resident in RAM while they're being edited but can be stored on cassette or disk for later use.

The Assembler is used to "assemble" (see ALWI) the source files in RAM. This consists of translating the symbolic source code representing Z-80 machine-language instructions into an "object file." The object file is largely machine language codes (see MLWI) with minor "header" information (see CFFT) that indicates the area of RAM to be loaded and other data.

The object file, when properly assembled, represents a machine language program that can be loaded under TRSDOS (see LMLD) or the cassette SYSTEM command (see LMFN) and executed. The object "code" may be anywhere from several instructions to thousands of instructions long and is dependent upon the application.

Before proceding, you must know something about Z-80 assembly language. See ALWI.

To Create a New Source Program:

1. While in the Command Mode, as indicated by the '*,' type I. This puts you into the Line Insert Mode, as shown in Figure S1EA-1, starting with line number 00100.

Figure S1EA-1 — *Editor/Assembler Display*

```
   TRS-8Ø Series 1 Disk Editor/Assembler Version 1.Ø
  (c) 1981 Tandy Corporation. All rights reserved.
    Dervived from original Tape/Editor/Assembler
  (c) 1978 Microsoft. Licensed to Tandy Corporation.

*1          CURSOR, WAITING FOR YOUR FIRST LINE
ØØ1ØØ  █    OF ASSEMBLY LANGUAGE
```

2. Enter your source lines one at a time. While entering your lines, you can use the Edit Mode subcommands. These are virtually identical to the BASIC Edit Mode subcommands described in EMBH and allow you to edit characters within a line. Each line is terminated with an ENTER.

3. When you have entered the last line, type BREAK to get back to the Command Mode. You can now list and correct the source lines, save the lines as a source file, or perform other Edit actions. We'll describe each in turn.

To List and Correct Lines and Perform Other Line Edit Actions:

Besides the Edit Mode subcommands that operate on characters within lines, you can also delete, modify, or add lines to the source file. All of these commands work while in the Command Mode, as indicated by the asterisk (*) prompt.

1. *To Edit any line* on a character basis, enter En ENTER, where n is the line number. To Edit line 1100, for example, you'd enter

```
E1ØØ
```

to get back into the Edit Mode described above.

2. *To Print (display)* the source lines, enter Pn:m, where n is the starting line number and m is the ending line number. To display lines 300 through 1060, for example, you'd enter

```
P3ØØ:1Ø6Ø
```

Display one line by a Pn. Display the Top (first) line by T or P#. Display the Bottom (last) line by B or P* Display the current line by P(period).

3. *To get a hardcopy listing* use the H(ardcopy) command for your system line printer in the same formats as 2. To print lines 400 through 450, for example, you'd enter

```
H4ØØ:45Ø
```

4. *To delete a line* or range of lines, enter Dn or Dn:m, where n is the starting line to be deleted and m is the ending line. The starting line must exist, otherwise you'll get an error message.

5. *To insert a line*, use the In command, where n is the line number of the insert line. The line number generally follows the line number immediately before the insert point. To insert a source line directly after line 1010 and before the following line 1020, you'd enter

```
I1Ø11
```

To insert a series of lines, use the I command. EDTASM will automatically renumber the lines as new lines are added. All lines will be inserted at the same point. If you'd like to avoid the renumbering, use the In,m form of the insert, where m is a line increment. If the existing lines increment by 10, for example, you'd be able to insert 9 lines between lines 1010 and 1020 without renumbering by

```
I1Ø11,1
```

6. *To renumber the lines*, use the Nn,m command, where n is the starting line number (often 100) for renumbering, and m is the increment (often 10).

7. *To replace a line* and continue in the line insert mode, use the Rn command, where n is the line number of the line to be replaced. The line will be replaced with the next entered line and from that point the operation will be identical to the line insert mode described in 5 above.

8. *To find a text string.* Use the Fxxxx command, where xxxx is a text string to be found.The F command is handy for locating lines by looking for "labels." If you want to look for a series of labels, enter Fxxxx and follow with F alone. The command will use the previous search string.

9. *To return to TRSDOS or Level II BASIC*: Enter Q for Quit.

10. *To find the amount of memory remaining*: Type M for Memory. EDTASM will indicate the bytes remaining.

To Write a Source File:

After you have massaged your source lines by the Line Edit and Character Edit commands above, you can either assemble the file (see below) or write out the source file to cassette or disk.

To write to cassette or disk:

1. Enter W name, where name is the name of the file. If no name is specified, the name NONAME (cassette) or NONAME/SRC (disk) will be used. If no disk extension is used, the extension /SRC will be used. If you do not want an extension on a disk file, enter a slash after the disk file name.

2. Cassette: You'll now see a READY CASSETTE message. Prepare the cassette and press any key. Both: cassette operations or disk writes will proceed until all the source lines are written.

3. After the write, the asterisk (*) prompt will be displayed.

To Load an Old Source File:

An old source file can be loaded for assembly or modification by this procedure:

1. While in the command mode (asterisk prompt), enter Lname.

2. If you are loading from cassette, EDTASM will prompt you to prepare the cassette. If you are loading from disk, EDTASM will load the source disk file. If you do not enter a name, either NAME (cassette) or NONAME/SRC (disk) will be used. If your disk file has no extension, use a slash after the name.

3. If you have an existing source file in memory, EDTASM will ask

```
TEXT IN BUFFER, CHAIN FILES?
```

If you answer Y, the new file will be added on to the end of the current file in memory, and the source lines will be renumbered. If you answer N, the new file will replace the source lines in memory.

To Assemble:

Once you have a good source file with no typographical errors and proper format (or even before, I don't care), you can assemble. Assembling translates the source code lines into equivalent machine language code (see ALWI). You can assemble with the following commands and options:

1. *To assemble without an object file* to cassette or disk:

A. While in Command Mode (*), enter A,NO.

B. You'll see a listing rapidly displayed on the screen.

C. At the end of the listing you'll see XX ERRORS, but it will go by too fast too observe.

D. You'll then see a "Symbol Table" display.

2. *To assemble without an object file and with no symbol table listing*:

A. Enter A,NO,NS.

B. You'll see a listing and error indication with no symbol table. Any errors will have been displayed too fast to catch.

3. *To assemble without an object file and to "wait on errors"*:

A. Enter A,NO,WE or A,NO,NS,WE, depending upon whether you want a symbol table listing (NS is No Symbol Table).

B. The display will stop as each assembly error comes up. Press any key to continue.

C. You'll get the error message at the end.

4. *To assemble to line printer without object*:

A. Use any of the commands above, but add ,LP to the end of the command line.

B. You'll get a simultaneous display and line printer listing.

5. *To get object output*:

If you use any of the above command sequences without the ,NO option, you'll get "object" output to either cassette or disk, with either a NONAME (cassette) or NONAME/CMD (disk) name. (Note that the disk extension is now "/CMD".) If you use the format A name,XX,XX,XX, where XX are options (switches), you'll get an object output to cassette or disk. The name used for disk will be name/CMD, unless you use a slash as the end character for the name.

6. *To load the object file and execute*, see LMFN or LMLD.

For more information on assembler operations, see:

ALWI for general information on assembly language

POZA for information on assembler pseudo-ops

AEDI for assembler errors

AEU1for assembler expressions

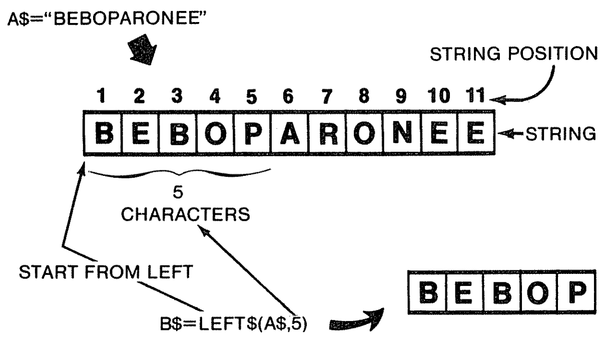
SACW
Strings, BASIC, Accessing Characters Within

Review SHTU if you're not very familiar with strings.

There are three basic ways to get characters within a character string: LEFT$, RIGHT$ and MID$. (If you have a Model I or III with Level I, however, these commands do not apply).

LEFT$ lets you obtain a chunk of the string starting from the leftmost character of the string, as shown in Figure SACW-1.

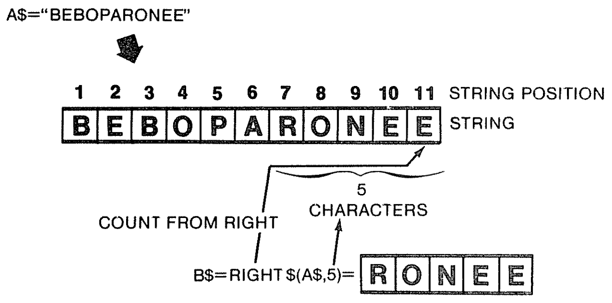
Figure SACW-1 – *LEFT$ Operation*



The format of LEFT$ is LEFT$(string name,n) where "string name" is the string variable name, and "n" is the number of characters in the chunk. If the string was A$="BEBOPARONEE", the command B$=LEFT$(A$,5) would set B$ equal to "BEBOP."

RIGHT$ lets you obtain a chunk of the string *ending* with the rightmost character, as shown in Figure SACW-2.

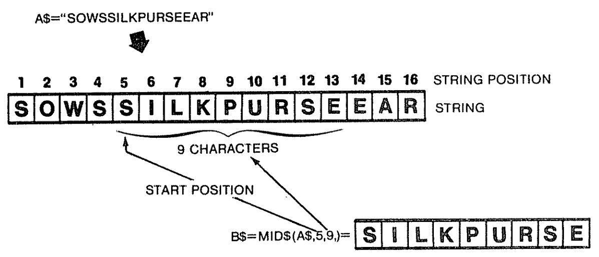
Figure SACW-2 – *RIGHT$ Operation*



The format of RIGHT$ is RIGHT$(string name,n) where "string name" is the string variable name, and "n" is the number of characters in the chunk. If the string was A$="BEBOPARONEE", the command B$=RIGHT$(A$,5) would set B$ equal to "RONEE."

MID$ lets you obtain a chunk of the string from the middle of the string, a so-called "substring", as shown in Figure SACW-3.

Figure SACW-3 – *MID$ Operation*


A$="SOWSSILKPURSEEAR"
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 STRING POSITION
S O W S S I L K P U R S E E A R STRING
9 CHARACTERS
START POSITION
B$=MID$(A$,5,9,)=
S I L K P U R S E

The format of MID$ is MID$(string name,p,n) where "string name" is the string variable name, "p" is the starting character (numbered from 1, the leftmost character), and "n" is the number of characters in the chunk. If the string was A$="SOWSSILKPURSEEAR", the command B$=MID$(A$,5,9) would get a "SILKPURSE" (B$) out of the surrounding "SOWS" and "EAR." Of course, A$ would still be equal to "SOWSSILKPURSEEAR".

In the examples above, we've used constants for "n" and "p", but these could also be variables or expressions, as in B$=LEFT$(A$,N) or B$=MID$(A$,(N-1),(M-2)).

These string functions are used to "strip" strings of character "fields." We might have one compact string representing last name, first name, and number of pirated programs, as in A$="CAPONE*AL*1233*"; the three fields could be obtained by clever use of the string functions (or even by not so clever use).

SBPD
Saving BASIC Programs on Disk

BASIC programs can be saved on diskette and reloaded at any time.

The two basic commands for doing this are SAVE and LOAD.

To save any BASIC program, do either a

```
SAVE "name"
```
or a
```
SAVE "name",A
```

The first command writes out the current program in user RAM as a disk file called "name." See FNMH for the formats of disk file names. At a minumum, any 1- to 6-character name starting with an alphabetic character will do nicely for a file name.

The second command writes out the current program as a disk file called "name." The file will be written out as an ASCII file (see AFWA) without the normal BASIC encoding of tokens (see TM13, TMTW, TBCC).

Writing takes anywhere from a few seconds to dozens of seconds, depending upon the length of the program.

To read in a previously written file do a

```
LOAD "name"
```

The BASIC file will be loaded from disk and stored, ready to RUN in RAM. The LOAD may be either of a BASIC file in ASCII, or a non-ASCII file.

To LOAD and RUN a BASIC program file do a

```
RUN "name"
```

The file will be loaded and immediately executed.

Two other options are LOAD "name",R and RUN "name",R.

The LOAD "name",R option clears all variables, will not close open disk files, and will execute the loaded BASIC program from the first line.

The RUN "name",R option *does not clear* variables, will not close open disk files, and will execute the loaded BASIC program from the first line.

All LOADs and RUNs can be used within a BASIC program. Use the LOAD "name",R to load one BASIC program from another where both share common disk files; use RUN "name",R to load one BASIC program from another where both share common variables and/or disk files. The latter technique is called "chaining" programs or "overlaying" one program with another, and is used to split a large application into several segments that will fit into memory at different times.

SBPT
Saving BASIC Programs on Cassette, All Systems

BASIC programs can be saved on cassette tape and reloaded at any time, except for Model II users. Cassette saves and loads can be used even in Disk BASIC, TRSDOS or LDOS.

The two basic commands for doing this are CSAVE and CLOAD.

To save any BASIC program, do either a

```
CSAVE
```

or a

```
CSAVE "name"
```

The first command writes out the current program in user RAM as a cassette file called "NONAME." The second command writes out the current program as a cassette file called "name." File names are 1 character long in the Model I/III and from 1 to 6 characters long in the Color Computer.

Writing takes anywhere from a few seconds to several minutes, depending upon the length of the program.

Model III users only: The file will be written at 1500 baud (see FBWT) unless you have answered the Cass? prompt on power up with "L", in which case it will be written at 500 baud. Use the 1500-baud mode for all CSAVEs as it is just as reliable as the 500-baud speed. The exception would be if you are creating Model I compatible cassette files, which load only at 500 baud.

Model I/III (except Level I) only: After the write, you may do a CLOAD? after rewinding the cassette and repositioning it before the cassette file. The CLOAD? will read in the file, but will not store it into user RAM. It serves only to compare the file on cassette with the program in RAM and verify that it is good. If the comparison is not valid, you can try rewriting the file. You should have little trouble on the Model III, but may have some problems on earlier Model Is. (As a matter of fact, you'll probably not use the CLOAD?, but just assume the file is good; as an alternative to CLOAD?, write another copy of the BASIC program on cassette after the first).

To read in the file, rewind the cassette and position the tape before the file you want to load (use the tape counter or audio output to do this). Now do either a

```
CLOAD
```

or

```
CLOAD "name"
```

to read in either the next file or the specified file called "name."

The file should take the same time to load as it did to write. You should see asterisks on the right-hand corner of the screen as the file loads. If there are one or more files before the file you've specified, you'll see the file name displayed in the upper-right hand corner of the screen.

If you experience problems in loading cassette files, see CTLC.

SCIB
String Concatenation in BASIC

String concatenation simply means that one string is appended to another. Suppose we had

```
100 A$="GOOD BOOKS FROM"
110 B$=" IJG"
```

We could concatentate the strings by

```
120 C$=A$+B$
```

to get C$="GOOD BOOKS FROM IJG". Big deal.. I mean the concatenation, not the books ...

SCM1
Screen Clear in BASIC, Model I/III - What's Used?

The CLS command clears the screen by storing all blank characters (**20H**). If you are going to do graphics work in assembly language, make certain that the screen areas involved are set to graphics blanks by first clearing the screen with POKEs or assembly language stores of 128 (**80H**).

SCOF
/CIM on Files, What Does It Mean?

The CIM extension on disk files is automatically generated from some TRSDOS commands (such as the Model I DUMP command) unless you specify another extension. It stands for "core image," an antiquated term that means that the data duplicates the contents of RAM. (Yep, sonny, back in the days of core memory, thutty years ago . . .)

SDFB
Sequential Disk Files, BASIC, Using

Sequential disk files are one of the 2 types of files (the other is "random," see RDFB) that Disk BASIC can use. Sequential files are easy to use, easy to decode on LISTings and other inspection methods, and easier to understand than other types of files. On the other hand, sequential files are just that — sequential. When you access sequential files you generally start at the beginning of the file and then go through from beginning to end to find the record you're looking for. This is fine for certain types of processing (merging alphabetized records in alphabetized files, for example) but time consuming for other operations (finding random records from a master file of part numbers, for instance).

Parts of a Sequential File: Sequential files contain "records." Each record may be of "variable length," although this is not absolutely necessary. There may be any number of records in a file, within the constraints of the disk data area available. Generally, a record is related to some organized unit of data. A record in an inventory file, for example, might have a part number, manufacturer's part number, description, number on hand, number ordered, and so forth.

General Flow for Creating a Sequential File: To create a sequential file, do this:

1. Execute a BASIC OPEN statement. The format of OPEN is

```
100 OPEN"O",1,"INVENTOR/APR"
```

The above example "opens" a file called INVENTOR/APR for sequential "O"utput from buffer 1.

There's nothing mysterious about OPENing a file. The operation just makes an entry in the disk directory of files, putting in the name, spec'ing it as a sequential file, and finding a vacant space on the diskette for the file to use. At this point the DOS does not know how large the file will be, and doesn't make any assumptions about the disk area required.

The buffer number is simply the buffer to be used to collect the data for the file. DOS does writes to the disk in 256-byte segments, rather than writing out each new piece of data, which would be inefficient; disk writes and reads take considerable amounts of time compared to other processing. The buffer number defines which one of several buffer areas are to be used with the file. One sector at a time is read into the buffer (or written from the buffer). Data is then taken a segment at a time from (or to) the buffer until the 256 bytes are exhausted; at that time another read (or write) is done.

2. Now that the file is OPENed, you can write to it. You may do as many writes as you want. Each write is very similar to a PRINT statement; as a matter of fact, a PRINT #1 will print to the sequential file in lieu of PRINTing on the display. The operation is very similar but the ASCII (character) data goes out to the disk file in lieu of PRINTing. The number for the PRINT corresponds to the buffer number.

Just as you could specify a list of PRINT items for display or line printer, you can specify a list of PRINT# items. Using the inventory example, you could store a part number, manufacturer's part number, description, number on hand, and number ordered, as follows:

```
100 OPEN"O",1,"INVENTOR/APR"
110 INPUT PN,MP,DE$,NH,NO
120 IF PN=-1 THEN GOTO 150
130 PRINT#1,PN,MP,DE$,",",NH,NO
140 GOTO 110
150 ...
```

We've done some really clever things here, like assigning names that match the item types, and so forth, in addition to looking for a part number of -1 to denote the end of entry.

The only thorny part in the write is the way string data is handled. As the file will use spaces between data items and the string data may contain spaces, a comma is used to separate the string from the other data items. The computereese word for this is "delimiter" — the comma "delimits" the end of the string.

3. When the terminating part number (-1) is entered, the file write is over. However, at this point there may still be data in the disk buffer, as the 256-byte buffer may not yet have been filled up from the last "dump" to disk. The CLOSE statement closes this buffer out by writing any remaining data to the disk file (in some circles this is called "flushing the buffer"); it also notifies the DOS disk file manage software that the disk file has been completely written and this results in the directory entry being completed, with the number of records in the file and other information.

The complete inventory program now looks like:

```
100 OPEN"O",1,"INVENTOR/APR"
110 INPUT PN,MP,DE$,NH,NO
120 IF PN=-1 THEN GOTO 150
130 PRINT#1,PN,MP,DE$,",",NH,NO
140 GOTO 110
150 CLOSE 1
```

The CLOSE statement "closed" the file associated with buffer number 1, our INVENTOR/APR program.

4. This general sequence of OPEN, PRINT#, and CLOSE is followed for every sequential file that is written.

General Format of Sequential Files: We can use the LIST command to see what the file data actually consisted of. Figure SDFB-1 shows a listing of a typical INVENTOR/APR file section.

This portion shows the entries:

```
1000,12345,32K rams half good,12,24
1010,876554,AK47 children's assault rifle,23,0
```

representing two inventory records for a typical Radio Shack store.

Why are there blanks (**20H**)? There are blanks because commas were used between the PRINT items. The data was written out exactly as it would have been to the display, with "tabs" (see CHTP), and those tabs resulted in blanks.

Note that there is a comma after each string in a record. This is the only comma appearing in the record and it is in there because we explicitly put it in. Every record is terminated by a **0DH** byte. This is a carriage return, and duplicates the carriage return that would be done for a PRINT line on the display.

Note also that the two records shown here are of different lengths. There are also some other interesting items, such as leading blanks before numeric items.

Obviously some space has been wasted here. Let's try again with semicolons in place of commas. We'll use this PRINT statement:

```
130 PRINT#1;PN;MP;DE$;",";NH;NO
```

After the same entries we have the two records shown in Figure SDFB-2.

Figure SDFB-1 – *Sample Sequential File*

```
ØØØØ:ØØ = 2Ø 31 3Ø 3Ø 3Ø 2Ø 2Ø 2Ø  2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø    1ØØØ
ØØØØ:1Ø = 2Ø 31 32 33 34 25 2Ø 2Ø  2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø    12345
ØØØØ:2Ø = 33 32 4B 2Ø 72 61 6D 73  2Ø 68 61 6C 66 2Ø 67 6F   32K rams half go
ØØØØ:3Ø = 6F 64 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø  2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø   od
ØØØØ:4Ø = 2C 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø  2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø    ,
ØØØØ:5Ø = 2Ø 31 32 2Ø 2Ø 2Ø 2Ø 2Ø  2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø    12
ØØØØ:6Ø = 2Ø 32 34 2Ø ØD 2Ø 31 3Ø  31 3Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø    24 . 1Ø1Ø
ØØØØ:7Ø = 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 38 37  36 35 35 34 2Ø 2Ø 2Ø 2Ø         876554
ØØØØ:8Ø = 2Ø 2Ø 2Ø 2Ø 2Ø 41 4B 34  37 2Ø 63 68 69 6C 64 72        AK47 childr
ØØØØ:9Ø = 65 6E 27 73 2Ø 61 73 73  61 75 6C 74 2Ø 72 69 66   en's assault rif
ØØØØ:AØ = 6C 65 2Ø 2Ø 2Ø 2C 2Ø 2Ø  2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø   le    ,
ØØØØ:BØ = 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 32 33  2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø         23
ØØØØ:CØ = 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 3Ø 2Ø  ØD                                Ø .
```

Figure SDFB-2 – *Semicolon Space Compression*

```
ØØØØ:ØØ = 2Ø 31 3Ø 3Ø 3Ø 2Ø 2Ø 31  32 33 34 35 2Ø 33 32 4B    1ØØØ  12345 32K
ØØØØ:1Ø = 2Ø 72 61 6D 73 2Ø 68 61  6C 66 2Ø 67 6F 6F 64 2C    rams half good,
ØØØØ:2Ø = 2Ø 31 32 2Ø 2Ø 32 34 2Ø  ØD 2Ø 31 3Ø 31 3Ø 2Ø 2Ø    12  24 . 1Ø1Ø
ØØØØ:3Ø = 38 37 36 35 35 34 2Ø 41  4B 34 37 2Ø 63 68 69 6C   876554 AK47 chil
ØØØØ:4Ø = 64 72 65 6E 27 73 2Ø 61  73 73 61 75 6C 74 2Ø 72   dren's assault r
ØØØØ:5Ø = 69 66 6C 65 2C 2Ø 32 33  2Ø 2Ø 3Ø 2Ø ØD            ifle, 23  Ø .
```

This time the records occupied much less space. The difference is because the semicolons eliminated the spaces between the items as they were PRINT #ed to the disk file. More on sequential file formats later in this procedure.

General Flow for Reading a Sequential File: To read a sequential file, do this:

1. Execute a BASIC OPEN statement. The format of OPEN is

```
1ØØ OPEN"I",2 ,"INVENTOR/APR"
```

The above example "opens" a file called INVENTOR/APR for sequential "I"nput from buffer 1.

The OPEN here is similar to the OPEN for creating a file. Here we know the name of the file we wish to OPEN, as it must exist to read it. The buffer used in the OPEN must be a second buffer if a Write File is also taking place. The DOS disk file manage software finds the given file name from the disk directory and locates its area on disk.

The buffer number is the buffer to be used to store disk file sectors of 256 bytes. DOS does reads to the disk in 256-byte segments, rather than reading each new piece of data, which would be inefficient. Data is accumulated in the 256-byte buffer until it is filled, and then a write is done to the next disk sector.

2. Now that the file is OPENed, you can easily read from it. You may do as many reads as you want. Each read is very similar to an INPUT statement; an INPUT #1 will read from the sequential file in lieu of INPUTting from the keyboard. The number for the INPUT corresponds to the buffer number.

Just as you could specify a list of INPUT items for keyboard input, you can specify a list of INPUT # items. The INPUT # items should correspond to the PRINT # items that were written out to the sequential file. You know this sequence beforehand, of course. Using the inventory example, you should input a part number, manufacturer's part number, description, number on hand, and number ordered, as follows:

```
1ØØ OPEN"I",2 ,"INVENTOR/APR"
11Ø INPUT# PN,MP,DE$,NH,NO
12Ø IF EOF(2) THEN GOTO 15Ø
13Ø PRINT PN,MP,DE$,NH,NO
14Ø GOTO 11Ø
15Ø CLOSE 1
```

We've used the same variable names as in creating the file.

3. The EOF statement checks for the "end of file" or EOF. The DOS disk file manage software knows when this EOF comes up as it reads in the file; it knows the number of records in the file from the disk directory entry describing the file. At this point there is no more data in the file buffer as there was in the Write File case. It is good programming practice, however, to CLOSE any buffer to release it to the system for possible other use.

4. This general sequence of OPEN, INPUT#, and CLOSE is followed for every sequential file that is read.

In the example above, the BASIC interpreter would read in an item at a time, looking for the proper delimiter between each data item in the disk file. It would look for spaces between numeric values and for a comma delimiter at the end of strings. For delimit rules, read on further in this procedure.

Method for Appending Data to an existing Sequential File: Suppose that you have a sequential file open with data in it. If you want to add data to it, you cannot simply do an OPEN"O", because that will OPEN the file for output starting at the first record. Instead, an OPEN"E" is done, which essentially says, "OPEN the file, find the last record, and get ready to add data at the end". Using our Radio Shack inventory program we'd have

```
100 OPEN"E",1,"INVENTOR/APR"
110 INPUT PN,MP,DE$,NH,NO
120 IF PN=-1 THEN GOTO 150
130 PRINT#1,PN;MP;DE$;",";NH;NO
140 GOTO 110
150 CLOSE 1
```

At the end of this session, the INVENTOR/APR file would have all input data appended at the end.

How to Find the End of a File: In the examples above, we used the EOF function to determine where the end of a file was. The general procedure is:

1. OPEN the file
2. Test EOF by IF EOF(n) THEN GOTO XXXX
3. Read the next record by INPUT #
4. Process the data
5. Loop back to 2.

EOF is a handy way to determine the end of a file, but not the only way. You could mark the EOF by writing out a last record with data items set to some unique value, such as -1 for a part number. You might also know beforehand that the number of records will be an exact number. These are perfectly logical ways to determine when the last record has been read, although EOF is more general case.

Further Format Considerations: Most of the trouble with sequential files comes from improper delimiting, if there is such a word (. . . there is now . . .), of data items in the records. There's no problem with numeric data items, but there is a problem with strings, because strings may contain some delimiting characters such as spaces or commas. Here are some rules for PRINT#s and INPUT#s:

1. If all of the data items to be written are numeric, no sweat, guys and gals. The numeric data items will have trailing blanks (ASCII **20H**) and they will delimit the data item from the next. Numeric data items, by the way, have either a leading blank or a minus sign, depending upon their value. Of course even here you must have the same number of INPUT # data items as you have data items in the disk file and in the same order. If you write out variables NH, OR, and DT for each record, you'd better read them back in in the same order.

2. A line of output is generated whenever a PRINT# is terminated by an ENTER in place of a comma or semicolon. A line is ended by a carriage return ASCII character (**0DH**) and generally represents one record. You can generate a record of 6 data items by many methods:

```
100 PRINT#1,A;
110 PRINT#1,B;
120 PRINT#1,C;
130 PRINT#1,D;
140 PRINT#1,E;
150 PRINT#1,F          (note no semicolon here)
1000 PRINT#1,A;B;C;
1010 PRINT#1,A;B;C   (note no semicolon here)
2000 PRINT#1,A;B;C;D;E;F
```

The three methods above generate the same record, 6 data items separated by spaces and the last terminated by a carriage return (0DH).

Actually, you would never really need a carriage return in the entire file. You could simply write out data items with semicolons. Carriage returns do lend themselves to using the LINEINPUT # command however, which helps in debugging (see below).

3. If you write out strings without quotes around them, a subsequent INPUT # with a string data item will start reading the string with the first non-blank. It will then read in all following characters as part of the string until:

A. A comma is found
B. The end of the file
C. A carriage return (generated by END OF LINE)
D. The 255th character

This poses several problems. If the string you've written has embedded commas, expect to get invalid reads, as the comma will act as a false delimiter. If you've written a string followed by a numeric data item, expect to get that data item and others included as part of the string. One way around this is by the next note.

4. You may enclose strings with double quotes. This involves doing something like

```
100 PRINT#1,CHR$(34);A$;CHR$(34);AS
```

In this example, the CHR$(34) supplies a double quote character in front and behind the string A$. A$ itself can contain anything — spaces, commas — anything but a double quote that is. The next double quote, carriage return, or end-of-file will delimit the string.

5. If a carriage return (ENTER) is preceded by a down arrow character, the carriage return is not taken as a delimiter, but as part of a string, or it is ignored for a numeric item.

6. Lots of rules, eh? One would think there would be an easier way to do all of this . . . Well . . . there isn't.

Using LINEINPUT #: LINEINPUT# reads in an entire line from a sequential data file. It stops only when it encounters a carriage return (0DH), the end of file, or the 255th character. All of the spaces, commas, double quotes and other anomalies we've been discussing above are included in the line.

LINEINPUT# is handy for "seeing what's out there," and one suspects it was included because everyone was having so darn much trouble with the delimiters in sequential files.

Another good use for LINEINPUT# is in processing BASIC, SCRIPSIT or other ASCII files. You can pretty well read in anything, although some of the non-ASCII data will result in graphics characters or weird screen actions if you try to print it without some processing. LINEINPUT # can be used to read in files, delete unwanted records, and write them back out again. The procedure goes something like this:

```
9Ø CLEAR 1ØØØ
1ØØ OPEN"I",1,"FILEA"
11Ø OPEN"O",2,"FILEB"
12Ø IF EOF(1) THEN GOTO 17Ø
13Ø LINEINPUT#1,A$
14Ø (process A$)
15Ø PRINT#2,A$
16Ø GOTO 12Ø
17Ø CLOSE 1,2
```

Using Multiple Buffers: You may OPEN as many files for input and output as you wish. Assign a different buffer number for each file in the OPEN. You may have up to 15 buffers in the Model I/II/III or Color Computer.

The number of buffers is defined on entry to BASIC.

Figure SDFB-3 shows a listing of a sequential file in which I've endeavored to show as many combinations of things as possible. You can use the DOS LIST command (see LDF1, etc.) to get your own listings of sequential files if you have problems.

Figure SDFB-3 – *Kitchen Sink Sequential File*

```
ØØØØ:ØØ = 2D 31 3Ø 3Ø 3Ø 2Ø 2Ø 31  3Ø 3Ø 3Ø 2Ø 2Ø 2Ø 2Ø 2Ø   -1ØØØ  1ØØØ
ØØØØ:1Ø = 2D 31 32 33 34 2Ø ØD 45  4D 42 45 44 44 45 44 2Ø   -1234 .EMBEDDED
ØØØØ:2Ø = 43 4F 4D 4D 41 2Ø 48 45  52 45 2C 57 49 4C 4C 2Ø   COMMA HERE,WILL
ØØØØ:3Ø = 5Ø 52 4F 44 55 43 45 2Ø  42 41 44 2Ø 52 45 53 55   PRODUCE BAD RESU
ØØØØ:4Ø = 4C 54 53 ØD 22 45 4D 42  45 44 44 45 44 2Ø 43 4F   LTS."EMBEDDED CO
ØØØØ:5Ø = 4D 4D 41 2Ø 48 45 52 45  2C 2Ø 4F 4B 22 ØD 45 4E   MMA HERE, OK".EN
ØØØØ:6Ø = 54 45 52 2Ø 5Ø 52 45 43  45 44 45 44 2Ø 42 59 2Ø   TER PRECEDED BY
ØØØØ:7Ø = 44 4F 57 4E ØA 41 52 52  4F 57 ØD                  DOWN.ARROW.
```

SDTD
SETting a Device to a Driver, Model I/III LDOS

Read LDIS on system devices. The SET command associates a logical device with a device driver. A device driver is a software program geared to a specific physical device or class of devices. The SET "links" the logical device with a physical device driver. The system may contain more than one device driver for certain logical devices. The *KI, or keyboard input logical device, for example, may use the KI/DVR driver, or it may use a user's custom-tailored driver. The KI/DVR is linked to *KI by

```
SET *KI TO KI/DVR
```

SET is used quite frequently as a way of initializing I/O. To set up the communications device, *CL, for example, you'd do a

```
SET *CL TO RS232/DVR (BAUD=...)
```

for the Model I.

The SET action is revoked by RESET (see RSLD).

SDWA
System Diskettes, What Are They?

"System" diskettes contain the disk operating system software. This software is loaded from disk as it is needed, by "overlays" into RAM. The operating system is too large to fit into RAM at one time; besides that, you'll need room for your program and variable storage. Whenever you are using TRSDOS, you must have a system diskette in drive 0 so that the operating system modules can be loaded as required. The other disk drives can contain "data" diskettes (see DDWA). Certain LDOS commands allow you to operate without a system diskette for short "utility" functions, such as COPYs, however.

SFLO
Strings, Finding Length of, BASIC, Most Systems

(Not applicable to Model I/III, Level I.)

Use the LEN command as in L=LEN(A$). The length of the string will be returned as an integer value from 0 through 255. Don't forget that a "null" string of 0 length is possible as in

```
100 A$=""
```

Null strings "allocate" the string space and the string name, but create a string of 0 length.

SHTC
Scrolling, How to Control in BASIC, Model III

Normally, the Model III scrolls all lines while in BASIC or other system programs. To "protect" the first through 7 lines from scrolling, do a

```
100 POKE 16916,N
```

where N is 1 through 7. To reset the scrolling of all screen lines, do a

```
100 POKE 16916,0
```

The protected area can be used for program messages, headings and other functions, and the remaining portion of the screen can display inputs and other processing.

After doing the POKE, the next lines to be written will still be written in the protected area until the last line of the screen is reached; from that point on, PRINTs will result in scrolling of the unprotected area only. An example:

```
100 POKE 16916,7:CLS
110 FOR I=0 TO 1000
120 PRINT I
130  NEXT I
```

Results in 0-6 in lines 0 through 6 and a continuous scroll in lines 7-15.

You can use this "one time" PRINT capability to write in the protected area (as for a table heading). You can also POKE into the protected area (see PPKU) or do a PRINT @ (HTDS) to the area. However, if you do a PRINT @, remember that it will set the "current" display position in the protected area, and subsequent PRINTs will be written normally until the end of the screen, at which point scrolling will be in the unprotected area only.

SHTO
Soldering, How to

Soldering is an easily-learned skill. The soldering we're talking about here is primarily soldering of connector pins, integrated circuit sockets and electronic components, — not those watering cans you made in high school shop.

1. Go to your nearest Radio Shack store and buy the following equipment:

   A. A 25- or 30-watt soldering iron with a pencil tip (64-2067).

   B. A solder iron holder (64-2078).

   C. A spool of 0.062 *rosin* core solder (64-002).

   D. Desoldering braid (64-2090).

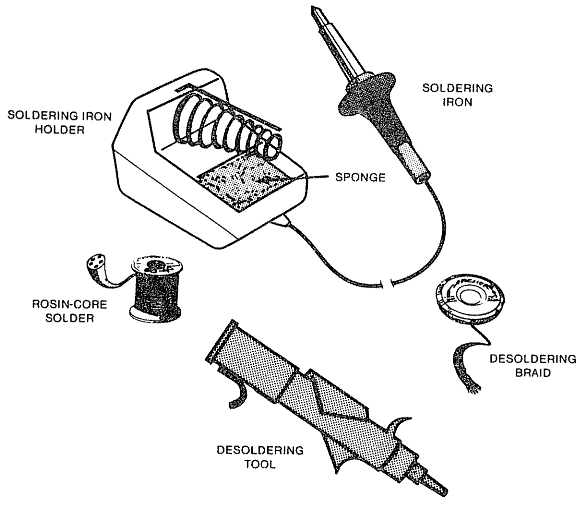
   E. Desoldering tool (64-2085).

These items are shown in Figure SHTO-1.

2. Put the iron in the holder and plug it in.

3. Take the sponge from the holder and wet it in cold water. It should be moist but not dripping.

Figure SHTO-1 – *Soldering Tools*

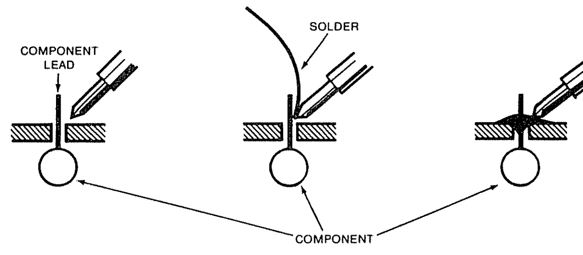


4. "Tin" the tip of the soldering iron by melting solder on it, rubbing the tip over the sponge, melting more solder, and so forth. When the tip is properly tinned it should look shiny.

5. To solder a component lead or pin, do the following:

   A. Hold the tip against the lead as shown in Figure SHTO-2.

Figure SHTO-2 – *Soldering*



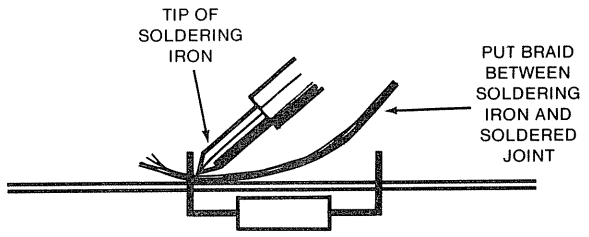
B. At about the same time, apply the solder near the junction of the tip and lead. Ideally, the heat from the lead should melt the solder, but usually it's a combination of heat from melted solder, the lead and the soldering iron tip.

C. Hold the iron a fraction of a second past the point at which the solder melts. Sorry, this comes with experience.

D. Look at the soldered junction. It should look like a somewhat shiny blob, with no "points" that formed when the iron was pulled away.

The soldering braid is used to remove solder from a joint, as shown in Figure SHT-3. Keep cutting off the braid as new solder melts up into the wires of the braid.

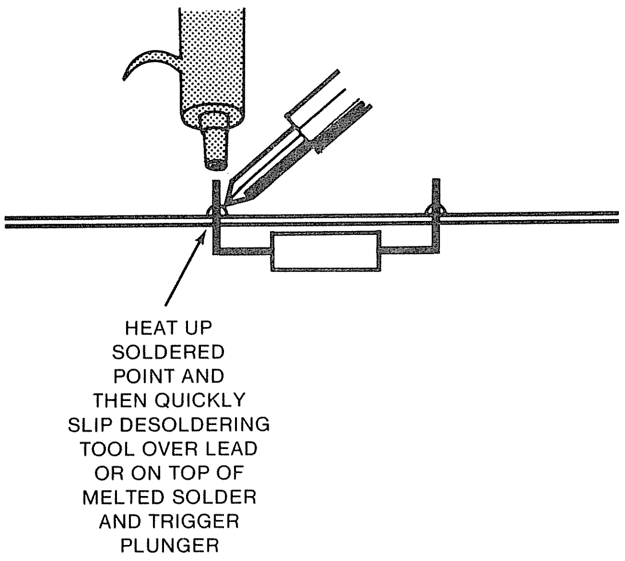
Figure SHTO-3 – *Desoldering Using Braid*



Another way to remove solder: Heat the soldered joint or connection and then "fling" the part downward. The solder will fly off. Best to do this away from expensive carpeting.

The desoldering tool is a suction device. Cock the gun by pulling back on the plunger. Do not point the gun at anyone. Remember, solder doesn't kill people, people kill people. Heat the junction as shown in Figure SHTO-4. Quickly remove the iron, put the tip of the tool over the solder, and press the trigger. You may have to repeat this several times.

Figure SHTO-4 – *Desoldering Using Solder Sucker*


HEAT UP SOLDERED POINT AND THEN QUICKLY SLIP DESOLDERING TOOL OVER LEAD OR ON TOP OF MELTED SOLDER AND TRIGGER PLUNGER

SHTU
Strings, BASIC, How to Use

Strings are groups of ASCII characters (see ADFW) that generally represent the letters A through Z, a through z, common special characters such as "&" and "#" and the digits 0 through 9. Often the string makes up a meaningful name or phrase, as in "ESCHER,M.C.", "INSERT DISKETTE", or "HOW MAY I HELP YOU?". The string may also be more abstract, though, as in "BBROYGBVGW", which represents the first letters of black, brown, red, orange, yellow, green, blue, violet, green , and white, the "resistor color code."

A string may also represent characters that are not "displayable" or are not ASCII characters, such as graphics characters (see GC13) or "control codes" (see ADFW).

You can see the advantage in using strings, as any program generally has to be able to display messages and to work with names, addresses, and so forth.

Strings are also called "string variables" because they are assigned a one- or two-letter name, such as A$, B$, A1$, or ZZ$. The first character of this name must be alphabetic; if there is a second character, it may be alphabetic or numeric. The dollar sign identifies the variable as a string.

If the string is known beforehand, it is a "string constant" and is defined in a statement such as 100 A$="INSERT DISKETTE". Double quotes enclose the string constant.

A string variable (not a constant) may be changed during program execution. Parts of strings may define other strings by using RIGHT$, LEFT$, or MID$ (see SACW). Strings may be appended to other strings by "concatenation" by using the "+" operator.

String constants or variables may be defined at any point in the program simply by naming an unused string variable and setting it equal to a string constant or expression, as in 2010 A$="ENTER NAME" and 3000 ZX$=A$+RIGHT$(BV$,5). Each string may be up to 255 characters in length. Use the BASIC CLEAR command (OSSH) to allocate enough space for string manipulation. A CLEAR 2000 early in the BASIC program sets aside 2000 bytes of space (or about 2000 string characters) for string manipulations.

A string of 0 length is called a "null string" and will not print or display.

Strings may also be used in arrays. Initially, a string in a variable or array is set to 0 length.

Model I/II/III only: use the DEFSTR command to define a "range" of string variables.

```
100 DEF A-G
```

for example, would define AB, AZ, DE, and G equivalent to AB$, AZ$, DE$, and G$.

SIPS
Semicolon, In BASIC PRINTs

A semicolon is used in a PRINT or LPRINT statement to avoid tabs, as is done in when commas separate PRINT items (CHTP). When a semicolon is used after an item in a PRINT list no spacing of any kind is done. The next item to be PRINTed or LPRINTed will be printed at the next print position.

Why do you get spaces between some items, then, wise guy?

Unfortunately, the semicolon can't control the output format of the data items. Here's the scam on the format of displayed or printed data item types:

1. Strings are printed with no leading or trailing blanks, so you have no problems there.

2. Positive numbers from integer, single-precision, or double-precision variables are printed with a leading and trailing blank.

3. Negative numbers from integer, single-precision, or double-precision variables are printed with a leading minus sign and trailing blank.

4. Numbers from numeric variables are not printed with leading zeroes or trailing zeroes, so they're going to be different lengths for the most part. (You'll never see *00*1.23 printed, for example, or 23.25*00*.)

If you want to use semicolons, but want things in nice, neat columns, or without leading or trailing blanks on numeric data, see CSNV for suggestions on what to do.

Remember that if a semicolon ends a PRINT or LPRINT, the next PRINT or LPRINT will start from the current "character position," which will not be a new line!

SNER
SN Error

Syntax Error. General BASIC response to garbled format such as

```
100 FRR I=1 TO 100        'should be FOR...
```

SPHT
Spooling, How to Use, Model I/III LDOS

Line printers operate at a hundred characters per second or so, but the TRS-80s operate at hundreds of thousands of characters per second. Much time is spent in typical systems "waiting for I/O." The program outputs a character and then waits for the line printer to respond with a signal that says "all right, send me the next character." In the space of time that the program is waiting for that response for the next character (1/100th second or so), the CPU could have executed 2000 instructions! That time is lost because the program is waiting in an I/O loop and dedicated solely to testing the printer "ready" status.

The Spooler in LDOS takes advantage of that high overhead time associated with I/O to overlap processing with I/O waits. It does it by "interrupts." Data that normally would go to the printer or other device (such as a slow-speed communications line) instead goes to a memory buffer. Periodically, the Spooler routine is entered and checks to see if there any characters to be output in the buffer. If there are, it outputs the next character and immediately releases control back to the current program for processing. Because the Spooler cannot exactly synchronize its speed with the I/O device it's servicing, the buffer may be empty when the Spooler is entered. On the other hand it may be full. In the latter case, the Spooler goes to an optional disk file to store the data.

The spooler distributes I/O evenly and avoids time consuming "I/O wait" conditions. The more I/O that your program has, the more you'll benefit from Spooling.

```
SPOOL *PR TO SPLFLE
```

sets printer spooling by linking the system print device to a disk file called SPLFLE. The memory buffer is automatically stored in high memory and is 1024 bytes long.

To use a larger memory buffer, do

```
SPOOL *PR TO SPLFLE (MEM=mm)
```

where mm is the number of bytes in K (1024) units to be used for the memory buffer.

Use the DISK option to allocate more than the default value of 1024 bytes of a disk file for spooling:

```
SPOOL *PR TO SPLFLE (DISK=mm)
```

where mm is the number of K (1024) bytes in the disk file.

Experiment with SPOOL without the MEM or DISK options to see how the memory sizes and disk affect your typical program. You'll notice a time lag between the time data would normally be printed and the actual printing during Spooling. This is normal.

To turn the Spooler off and reset the device spec, use

```
SPOOL *PR (OFF)
```

SPHU
Single-Precision Variables in BASIC, How to Use

(Does not apply to Model I/III, Level I, or Color Computer).

If you do not explicitly define a variable as integer (see IVHU), double-precision (see DPHU) or string (see SHTU), then it is a single-precision variable.

Single-precision variables take up 4 bytes of RAM storage, have 7 decimal digits of accuracy and can express a wide-range of numbers by "floating-point" representation.

A range of single precision variables may also be specified by the DEFSNG command. DEFSNG A-G, for example, specifies all variables starting with an A through G as single precision; AS, FD, and GG would be single precision in this case.

To define a variable as single precision after a DEFDBL, DEFINT, or DEFSTR (which define a "block" of variable names as double-precision, integer, or string variables), use the suffix "!". For example, in the following code, A! is a single precision variable:

```
100 DEFDBL A-G
110 A!=123.456
```

SRBH
SET/RESET in BASIC, How to Use, Model I/III

Must you? I mean, really, how could you use something so slow? See GHS1 for descriptions of high-speed graphics. SET/RESET, however, *are* good for plotting data and producing point graphs — any situation where there is no predictable horizontal or vertical line, pattern, or figure.

SET/RESET and their comrade, POINT, are referenced to graphics blocks. If you don't know what I'm talking about, read GC13 first.

Each character position has six subdivisions, two columns and three rows, as shown in Figure GC13-1. The total number of blocks across, therefore, is 128, numbered 0 through 127. The total number of blocks down is 48, numbered 0 through 47. From here on in, forget about character positions and think in terms of x and y graphics references.

The formats for the commands are

```
SET (X,Y)
RESET (X,Y)
POINT (X,Y)
```

SET sets the specified block to on, or "white." RESET resets the specified block to off, or "black." POINT returns a value indicating whether a block is on (-1) or off (0). And that's about all there is to it; convert x/y coordinates to the Model I/III coordinate system where X=0 is at the left and Y=0 is at the top. A typical plot of a sine wave is shown below:

```
100 'PLOT SINE WAVE
110 CLS                          'CLEAR SCREEN
120 FOR ND=0 TO 359              '0 TO 359 DEGREES
130 RD=ND/(360/(2*3.14159))      'FIND RADIANS
140 X=128*(ND/360)               'FIND X DISTANCE
150 YV=SIN(RD)                   'FIND Y VALUE +1 TO -1
160 Y=24-YV*23                   'FIND Y VALUE 0 TO 47
170 SET (X,Y)                    'SET POINT
180 NEXT ND                      'NEXT DEGREE
190 GOTO 190                     'KEEPS DISPLAY
```

SSA3
Sorting a String Array in BASIC, Model III TRSDOS, Model I/III LDOS

Easy. Assume that in a one-dimensional string array (see SHTU) you have a set of unsorted strings. To reshuffle the strings and arrange the array so that it contains sorted strings from "lower weight" to "higher weight" (ascending sequence), execute

```
CMD"O",size of array,array name(Ø)
```

If the array is 100 elements for example, you might have something like this:

```
1ØØ DIM A$(99)           'dimension array
    .
    .
    .
1ØØØ CMD"O",1ØØ,A$(Ø)    'sort array
```

The array would be sorted on the basis of the "ASCII" (see ADFW) values of the strings. By a fortunate coincidence, ASCII codes increase in value for increasing alphabetical order, so the array would be in alphabetical sequence. Although the normal case would involve only text strings, control codes, special ASCII characters such as "#" or "." or graphic codes would also be sorted on the basis of their numerical values.

The general case format of the sort command is:

```
CMD"O",number of elements to sort,string array(start)
```

If you wanted to sort less than the total string array, you could specify the starting element and the number of elements to sort. In the example above, if you wanted to sort the 10 elements from A$(10) through A$(19), you'd specify

```
CMD"O",1Ø,A$(1Ø)
```

and the sort would be done.

The array strings can be of mixed length, just as you might suspect. In this case "EEEEE" would be sorted before "EEEEEE".

Be aware that ZZZZ appears before Zzzz and Zaaa! Any upper case character is lower in "weight" than any lower case character (see ADFW) and will appear first.

The CMD"O" can be used in the command mode, or in a BASIC program statement.

SSFA
String Space, Finding Amount Left

(Not applicable to Color Computer or Level I Systems). FRE(A$) returns the amount of free string space. Doing a 100 PRINT FRE(A$), for example, will print the free string space remaining. The string space area is established by a CLEAR statement (see OSSH). As strings are manipulated, this string space is used up; FRE allows you to check string space availability either within a program or in the command mode to see how much space is left.

STAC
Stack, Description

The stack area is an area set aside for machine language programs such as the BASIC interpreter. It is used constantly to store return addresses, temporary data, and interrupt addresses.

In normal BASIC use, forget about the stack; you needn't even be concerned that a stack exists.

If you are running machine language programs embedded in BASIC or other types of machine language programs, your instructions may refer to the stack, as in PUSHes or POPs (PSH,PULS in Color Computer). You don't have to *establish* the stack, however. The stack is almost always there if you are coming from BASIC or DEBUG, and there's more than enough room to handle your puny PUSHes of data, although I don't mean to denigrate your code. As a matter of fact, if you have entered your machine language code by a USR call, then you *must not* define the stack area (unless you restore the stack pointer prior to returning to BASIC).

The only time you would want to define the stack area is when you have a large assembly language program that runs independently of BASIC or other system programs. Then you would use an "LD SP,0C000H" (LDS in Color Computer) or similar instruction to set the "top of stack."

The stack builds down 2 bytes at a time for each PUSH (PSH) or CALL (BSR,JSR), and resets 2 bytes at a time for each POP or RET (PUL or RTS).

The stack area in BASIC is located directly below the area set aside for "string storage." See the memory map for your system in MMM1, MMM2 or MMCC (I/III, II, or Color Computer respectively).

STER
ST Error

String formula too complex. You've done it — you've finally given the BASIC interpreter something it couldn't handle, and it replies with an apologetic "ST."

Break it up into simpler terms, and feed it back to that wimp!

STRD
STRING$ Command, BASIC, Most Systems

(Not applicable to Model I/III, Level I).

The STRING$ command is a handy command to use for making up variable length strings that contain the same character — such strings as "################," "1111111111", or the popular "*****************." The format of STRING$ is

```
STRING$(n, "char")
```

where n is a number from 1 through 255, and char is the character to be repeated. A CHR$ command can be used to repeat non-ASCII data (see CUSE). The line

```
1ØØ A$=STRING$(2ØØ,"*")
```

generates a string of 200 asterisks, and sets string A$ equal to the asterisk string.

STWD
/TXT on Files, What Does It Mean?

The TXT extension on disk files means that the file is a "text" file consisting only of ASCII characters with each line terminated by a carriage return (ODH). See AFWA.

You can use the LINEINPUT# command (see SDFB) to read in a line at time, list the files on the screen or line printer, or simply sit there in rapture.

SUBB
Subroutines, BASIC, All Systems

What's a subroutine? Anything you want it to be, ranging from one BASIC command to hundreds. A subroutine is a set of BASIC code that has a predefined function and that can be used to replace two or more sets of identical code in your BASIC program. Using a subroutine in place of the code saves memory, as the subroutine will be at only one point in the BASIC program, rather than many. Most subroutines perform fairly specific actions, but any time code is duplicated at least once, you could use a subroutine to save memory.

A subroutine is called by a GOSUB, which marks the point at which the call was made. A RETURN command is the last statement in a subroutine and causes a return back to the next statement after the GOSUB. Subroutines may be "nested" in as many levels as necessary — a subroutine may call another subroutine, which may call another, and so forth.

Figure SUBB-1 shows typical subroutine action.

Figure SUBB-1 – *Subroutine Action*

```
1ØØ 'PLOTTER 1
1Ø1 LPRINT CHR$(18)
1Ø2 LPRINT"I"
11Ø CLS
12Ø LINE INPUT "COMMAND ";CM$
13Ø IF CM$="BOX" THEN GOTO 1ØØØ
14Ø IF CM$="ROWS" THEN GOTO 2ØØØ
15Ø IF CM$="COLS" THEN GOTO 3ØØØ
155 IF CM$="PSIZE" THEN GOTO 4ØØØ
16Ø IF CM$="ROWTXT" THEN GOTO 5ØØØ
17Ø IF CM$="VBKT" THEN GOTO 6ØØØ
18Ø IF CM$="VBKTT" THEN GOTO 7ØØØ
999 STOP
```

```
1ØØØ 'DRAW BOX (RECTANGLE OR SQUARE)                    "CALLS" SUBROUTINE
1ØØ1 INPUT SH,SV                                        AT LINE 1ØØØØ TWICE
1Ø1Ø ZA=SH:GOSUB 1ØØØØ:SH=ZA:ZA=SV:GOSUB 1ØØØØ:SV=ZA
1Ø2Ø ZB=SH:GOSUB 1Ø1ØØ:ZC=-SV:GOSUB 1Ø2ØØ:ZB=-SH } SUBROUTINE
     :GOSUB 1Ø1ØØ:ZC=SV:GOSUB 1Ø2ØØ               } CALLS TO
1Ø3Ø GOTO 12Ø                                       SUBROUTINES
2ØØØ 'DIVIDE LAST FIGURE INTO ROWS                  @ LINES 1Ø1ØØ,
2Ø1Ø INPUT NR                                       1Ø2ØØ, 1Ø2ØØ
2Ø2Ø RV=SV/NR
2Ø3Ø FOR I=1 TO NR-1
2Ø4Ø ZD=-RV:GOSUB 1Ø31Ø:ZB=SH:GOSUB 1Ø11Ø:ZE=-SH:GOSUB 1Ø41Ø
```

SUBP
Speeding Up Your BASIC Programs

There are a number of ways to speed up your BASIC programs. I won't rank them in their order of importance, but I will indicate what approximate effect they have on speed:

1. (Model I/II/III only.) Whenever possible, substitute integer variables for single-precision variables. Integer variables are denoted by a percent suffix, such as A%, and can hold values from -32,768 through +32,767. If you know that certain variables will not need to hold values out of this range, redefine the variables as integer. Great effect on speed.

2. Delete all REM or ' lines from your program. Delete all ' type remarks in a multi-statement line, such as

```
1ØØ A=127   'initialize variable A
```

Minor effect on speed.

3. Combine statements into multi-statement lines by using a semicolon. For example, convert

```
1ØØ A=127
11Ø B=23
```

to

```
1ØØ A=127: B=231
```

Moderate effect on speed.

4. Substitute variables in place of constants. For example, define pi once, as PI=3.14159, and then use the variable PI, rather than a constant. This eliminates time-consuming conversions, as BASIC can simply access the variable. Minor effect on speed.

5. Define commonly used variables first. BASIC searches for the variable reference from a variable list built up as each variable is defined. The list is searched from beginning to end. Defining commonly used variables early in the program will result in faster access. Define these variables in the first few program lines. Moderate effect on speed.

6. Put commonly used subroutines at the beginning of the program. When a subroutine is called, the BASIC statements are searched for the subroutine line number from beginning to end. Putting the subroutine at the beginning of the program will result in faster access. Moderate to great effect on speed, depending upon subroutine use.

7. Do not recompute expressions when possible. For example, the two statements

```
1ØØ  A$=RIGHT$(B$,5)
11Ø  C$=RIGHT$(B$,5)+B$
```

could be redefined as

```
1ØØ  A$=RIGHT$(B$,5)
11Ø  C$=A$+B$
```

Minor effect on speed.

8. Use POKE graphics or string graphics in place of SET/RESET. See GHS1 (Model I/III). Great effect on graphics speed.

9. Recode slow routines in assembly language. Assembly language may be easily incorporated into BASIC programs (see ALWI); the catch is that assembly language is horrendous to learn, compared to BASIC. Assembly language routines may run 300 timees as fast as the BASIC counterpart. Good candidates for assembly language: sorts and any extended "number-crunching" or heavy processing. Greatest effect on speed.

SUCG
Setting up for Color Computer Graphics in BASIC

SCREEN selects the type of screen and color set. The type parameter is 0 for text screen or 1 for graphics page. The color set code is either 0 or 1. SCREEN enables you to switch back and forth between the text screen at $400 and the "current" graphics page. Don't forget that the text and graphics screens are separate entities. The color set applies to the text screen as well; standard text display is black on green, but color set 0 selects red on orange.

PCLEAR reserves from 1 to 8 graphics pages. If PCLEAR is never used, 4 graphics pages are reserved automatically. Any remaining graphics pages are released to the common pool of RAM for variables, etc. Note that no action is taken in *clearing* the pages. This simply sets an internal pointer to the location after the last graphics page; this will be the start of the BASIC variable area.

PMODE selects a graphics mode of 0 through 4 and specifies the starting graphics page number of 1 through 8. PMODE may be used at any time to change the graphics resolution. PMODE is normally used early in a program to select the resolution and starting graphics page. It can also be used at any time to select a new graphics page. You might have one display starting on graphics page 0 and a second starting on graphics page 4, for example.

PCOPY copies the data from one graphics page to another. This is a convenient way to duplicate a graphics page without having to do a series of PEEKs and POKEs. More than one PCOPY might have to be done to duplicate a given display, as certain modes require more than one page.

PCLS is the Extended Color BASIC equivalent of the Color BASIC CLS. It clears the current graphics screen (which may be more than one page) to a given color.

COLOR selects the foreground and background color. The background color is the primer for the canvas. The foreground color is on the paint brush. This is important as some commands cannot specify a color for the graphics action. (See GMIC for colors.)

The normal sequence before starting any graphics would be something like

```
100 PMODE 4,1  'set mode, starting page
110 SCREEN 1,1 'select graphics, color set 1
120 PCLS 5     'clear screen with buff
```

SVDG
Setting the VDG Modes in the Color Computer Outside of BASIC

You can easily bypass the BASIC interpreter to set the VDG modes by POKEs from BASIC or your own assembly language program. Why would you what to do this? First of all, because you're a perverse sort and hate to be led around through the nose by the BASIC interpreter. Secondly, because you can now set some of the unimplemented VDG modes to see what they look like. Thirdly, you can implement all VDG modes even with Color BASIC (OK buddy - I'm booking you for a 2716, failure to buy Extended Color BASIC . . .)

Read GPAR to find out about the graphics architecture of the Color Computer.

The VDG modes are controlled by the three SAM outputs (see GPAR), and 5 PIA $FF22 bits, as shown in Figure SVDG-1. In addition, the most significant bit of the RAM byte determines whether the data is text or graphics in some modes.

To set any mode, output the proper configuration to the SAM by POKEing into addresses $FFC0 through $FFC5 as described above. Next, set the 5 PIA bits, carefully retaining the least significant three bits. In some cases, bit 3 of the PIA determines the color set, 0 or 1. You should see the data on the screen change to represent the mapping, colors, and area of the VDG mode you have set.

Unimplemented Semigraphics Modes: The three unimplemented "true" (not Semigraphics) modes are 64 by 64 in four colors, 128 by 64 in two colors, and 128 by 64 in four colors, as shown in Figure SVDG-2. These use the same memory mapping as the other graphics modes and are simply lower-resolution modes that save memory. I don't think that we're missing much by not having these modes included as part of the Extended Color BASIC. You be the judge.

Semigraphics 6, 8, 12, and 24 Modes: If you loved the graphics mapping of the TRS-80 Model I/III, you'll be estatic over the Semigraphics 6, 8, 12, and 24 modes. These modes offer two things: lots of colors and horrendous memory mapping.

The 6, 8, 12, and 24 suffix comes from the number of elements into which each character position is divided. The basic character position is 8 by 12 pixels; the 6, 8, 12 and 24 modes divide this basic unit into two columns of 3, 4, 6 or 12 rows, as shown in Figure SVDG-3.

The Semigraphics 6 mode is the easiest. There are 2 colors in this mode, controlled by the two most significant bits in each memory byte. The six elements of the character position are turned on or off to this color by a 0 or 1 bit in the remaining 6 bits of the RAM byte, as shown in Figure SVDG-4. This means that each consecutive byte in RAM controls one entire pixel position, as shown in the figure. A total of 512 bytes therefore control the display.

The Semigraphics 8 mode is a little harder. It has eight colors; each row is programmable as to color, as shown in Figure SVDG-5. The remaining halves of the four bytes control the on/off status of the elements. This mode requires four consecutive bytes in RAM to control one character position, making the total number of bytes required 2048.

The Semigraphics 12 mode is harder still. Colors in each row are controlled by three bits in six bytes as shown in Figure SVDG-6. The remaining halves of the bytes control the on/off status of the elements. This mode requires six consecutive bytes in RAM to control one character position, for a total of 512*6 or 3072 bytes.

The Semigraphics 24, or as it is called in the halls of Two Tandy Center, the "Jumbo" mode, is shown in Figure SVDG-7. Here again, color for each row is controlled by a separate byte, with the remaining portion of the byte controlling the on/off status.

The Semigraphics 6, 8, 12 and 24 suffer from lack of horizontal resolution. Resolution is only 64 elements, and each set of two must be the same color. On the other hand, they offer 8 colors in 12,288 pixels, which you certainly will not find in the other graphics modes.

If you are would like to use these modes, one of the biggest drawbacks would be the complicated algorithm to set or reset an element. Don't forget that you are working outside of the bounds of the BASIC interpreter, and will have to calculate the RAM byte and bit for each element. This will slow down the display operations considerably. It appears that these modes would be most useful for assembly language subroutines where the set/reset action could be done at acceptable rates.

Figure SVDG-1 – *VDG Mode Control*

SAM BITS			PIA & FF22 BITS					DATA BITS		MODE
V2	V1	VØ	7	6	5	4	3	7	6	
Ø	Ø	Ø	Ø	X	X	Ø	C	Ø	Ø	ALPHANUMERIC
Ø	Ø	Ø	Ø	X	X	Ø	C	Ø	1	ALPHANUMERIC INVERTED
Ø	Ø	Ø	Ø	X	X	Ø	X	1	X	SEMIGRAPHICS - 4
Ø	Ø	Ø	Ø	X	X	1	C	1	X	" - 6
Ø	1	Ø	Ø	X	X	Ø	X	1	X	" - 8
1	Ø	Ø	Ø	X	X	Ø	X	1	X	" - 12
1	1	Ø	Ø	X	X	Ø	X	1	X	" - 24
Ø	Ø	1	1	Ø	Ø	Ø	C	X	X	64 X 64, 4 COLOR
Ø	Ø	1	1	Ø	Ø	1	C	X	X	128 X 64, 2 "
Ø	1	Ø	1	Ø	1	Ø	C	X	X	128 X 64, 4 "
Ø	1	1	1	Ø	1	1	C	X	X	128 X 96, 2 "
1	Ø	Ø	1	1	Ø	Ø	C	X	X	128 X 96, 4 "
1	Ø	1	1	1	Ø	1	C	X	X	128 X 192, 2 "
1	1	Ø	1	1	1	Ø	C	X	X	128 X 192, 4 "
1	1	Ø	1	1	1	1	C	X	X	256 X 192, 2 "

SET THESE BY REFERENCE TO GPAR

RETAIN BITS 2 - Ø!

THESE BITS ARE FROM DATA IN VIDEO RAM

X = DON'T CARE
C = COLOR SET, Ø OR 1

Figure SVDG-2 – *Unimplemented Graphics Modes for Extended Color BASIC*

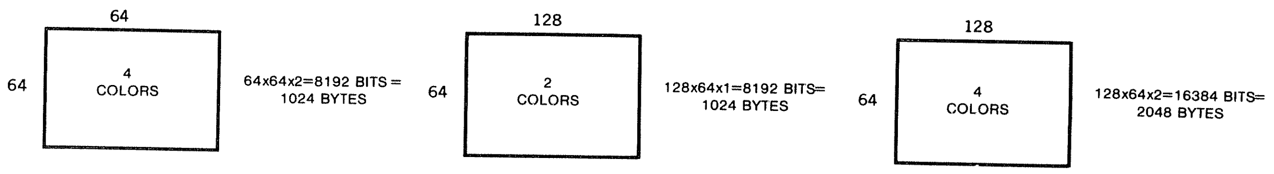

64
64
4 COLORS
64x64x2=8192 BITS = 1024 BYTES
128
64
2 COLORS
128x64x1=8192 BITS= 1024 BYTES
128
64
4 COLORS
128x64x2=16384 BITS= 2048 BYTES

Figure SVDG-3 – *Semigraphics 6, 8, 12, and 24 Character Position Division*

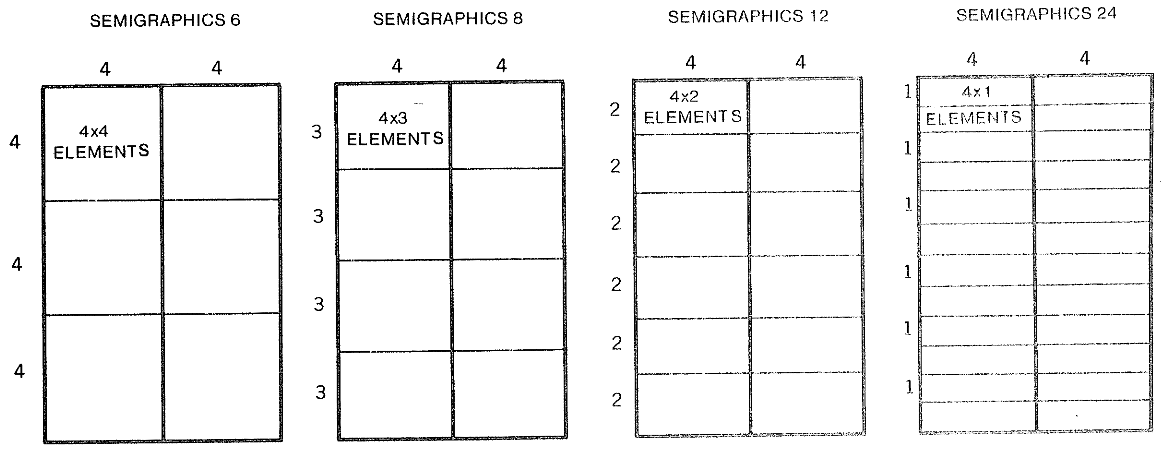


Figure SVDG-4 – *Semigraphics 6 Mapping*

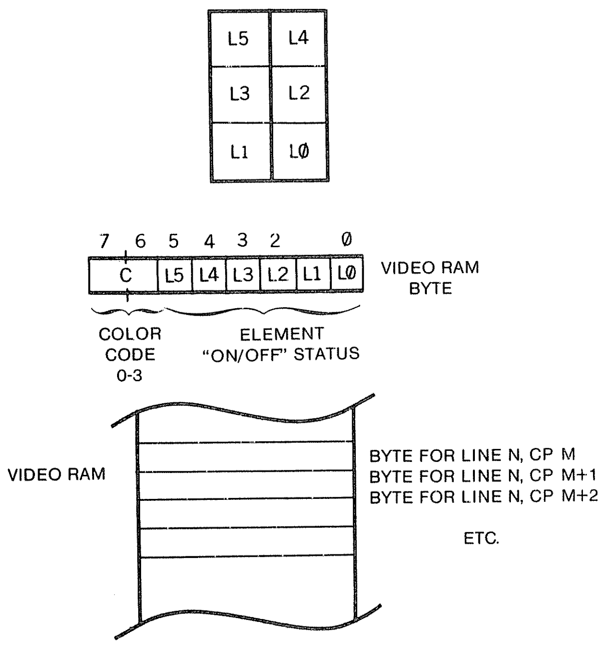


Figure SVDG-5 – *Semigraphics 8 Mapping*

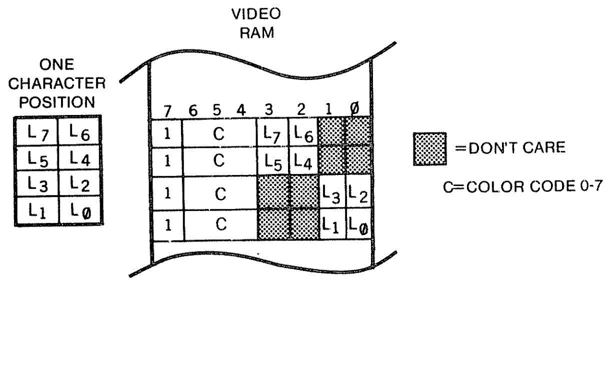


Figure SVDG-6 – *Semigraphics 12 Mapping*

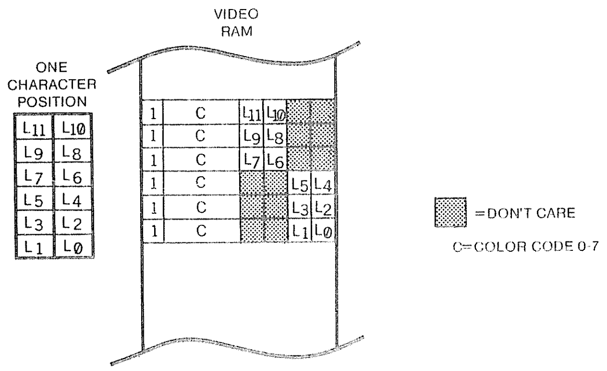
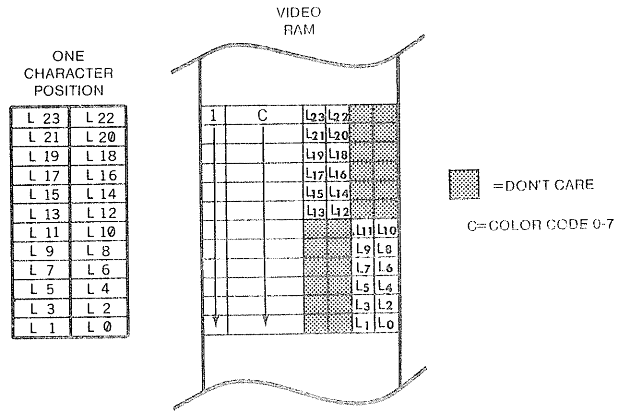


Figure SVDG-7 – *Semigraphics 24 Mapping*



SWHU
Slit and Wrap, How to Use

Don't. I cannot recommend this method of wire wrap to the novice. I have not had good luck with it and several experienced friends feel the same way. Typically there are 2 bad wraps per several dozen, and high resistance readings. Use ordinary wire-wrap instead (see WHHT). If you must use this method, there are detailed instructions with the Slit and Wrap tool.

TBCC
Tokens in BASIC, Color Computer

Color Computer BASIC and Extended BASIC use a special form of encoding for BASIC commands. BASIC commands such as LIST, NEW, FOR, RETURN and all others are not represented by their ASCII (see ADFW) equivalent text, but as one-byte or two-byte codes. This decreases the amount of memory storage required for BASIC programs. Tokens are always used in in-memory storage of BASIC programs, but BASIC programs may optionally be stored on disk as ASCII files (see AFWA), with "straight text" and no tokens.

The token codes used in both Color BASIC and Extended Color BASIC are shown in Table TBCC-1 below. Some are one-byte codes and some are two bytes, with the first byte equal to 255 (**0FFH**), as shown.

Table TBCC-1 – *Color Computer BASIC Token Codes*

Color BASIC One-Byte Tokens:

FOR	128
GO	129
REM	13Ø
'	131
ELSE	132
IF	133
DATA	134
PRINT	135
ON	136
INPUT	137
END	138
NEXT	139
DIM	14Ø
READ	141
RUN	142
RESTORE	143
RETURN	144
STOP	145
POKE	146
CONT	147
LIST	148
CLEAR	149
NEW	15Ø
CLOAD	151
CSAVE	152
OPEN	153
CLOSE	154
LLIST	155
SET	156
RESET	157
CLS	158
MOTOR	159
SOUND	16Ø
AUDIO	161
EXEC	162
SKIPF	163
TAB(	164
TO	165
SUB	166
THEN	167
NOT	168
STEP	169
OFF	17Ø
+	171
-	172
*	173
/	174
^	175
AND	176
OR	177
>	178
=	179
<	18Ø

Color BASIC Two-Byte Tokens
First Byte=255, followed by:

SGN	128
INT	129
ABS	13Ø
USR	131
RND	132
SIN	133
PEEK	134
LEN	135
STR$	136
VAL	137
ASC	138
CHR$	139
EOF	14Ø
JOYSTK	141
LEFT$	142
RIGHT$	143
MID$	144
POINT	145
INKEY$	146
MEM	147

Extended Color BASIC One-Byte Tokens:

DEL	181
EDIT	182
TRON	183
TROFF	184
DEF	185
LET	186
LINE	187
PCLS	188

PSET	189
PRESET	19ø
SCREEN	191
PCLEAR	192
COLOR	193
CIRCLE	194
PAINT	195
GET	196
PUT	197
DRAW	198
PCOPY	199
PMODE	2øø
PLAY	2ø1
DLOAD	2ø2
RENUM	2ø3
FN	2ø4
USING	2ø5.

Extended Color BASIC Two-Byte Tokens. First Byte = 255, followed by:

ATN	148
COS	149
TAN	15ø
EXP	151
FIX	152
LOG	153
POS	154
SQR	155
HEX$	156
VARPTR	157
INSTR	158
TIMER	159
PPOINT	16ø
STRING$	161

TCMH
32-Character Mode, How to Switch to in BASIC, Model I/III

1. While in BASIC, press SHIFT, right arrow simultaneously.

2. Execute a PRINT CHR$(23) command while in BASIC command mode or in a program.

The resulting display will be missing characters. Clear the screen and reLIST or rePRINT the program or data to get a valid display.

To switch back to 64-character mode, press CLEAR or execute a CLS command, either in command mode or in a program. This will clear the screen and reset the cursor to the upper left-hand corner (the "HOME" position).

TCML
Transferring Control to A Machine Language Program from BASIC, Model I/III Non-Disk

If there is a valid machine language program in RAM, you may transfer control to it by entering the SYSTEM mode of BASIC (see LMFN) by

```
SYSTEM ENTER
XX/MMMMM ENTER
```

where MMMMM is the starting address of the machine language program.

When is the program intact and when not? There's no easy way to tell. If the program is in high memory, and you have protected that memory area (see PROT), then the program should be fine. If the program is unprotected and at some intermediate point in memory, than it may be low enough so that it wasn't clobbered by the stack or string storage area (see MMM1) and high enough so that BASIC variables and arrays did not eat it up. Good luck!

TCNU
Two's Complement Numbers, Using

Two's complement numbers are used in Z-80 and 6809E assembly language programs on the TRS-80s and are also the internal form of "integer" and other variable types in BASIC.

If a binary number is in two's complement form, it may represent a positive or negative number. If a number is not in two's complement form, it is an "absolute" form, and represents only a positive value.

Let's take the case of a 16-bit binary value. The 16-bit binary value may represent the memory addresses from 0000000000000000 through 1111111111111111 (65,535) and be in absolute form. To convert between this form and hexadecimal or decimal, follow the procedures in CBBH and CFBD.

The 16-bit binary value may represent a BASIC integer value and be in two's complement form. (How can you tell when it is absolute and when it is two's complement? You can't tell by looking at the number. You must know which form is being used and act accordingly). To convert a

16-bit binary value in two's complement form, do the following:

1. If the number is 00000000 through 11111111, look in Table TCNU-1 to find the equivalent decimal value.

Table TCNU-1 – *Two's Complement Numbers*

DEC	BINARY	DEC	BINARY	DEC	BINARY	DEC	BINARY
-1	11111111	-33	11011111	-65	10111111	-97	10011111
-2	11111110	-34	11011110	-66	10111110	-98	10011110
-3	11111101	-35	11011101	-67	10111101	-99	10011101
-4	11111100	-36	11011100	-68	10111100	-100	10011100
-5	11111011	-37	11011011	-69	10111011	-101	10011011
-6	11111010	-38	11011010	-70	10111010	-102	10011010
-7	11111001	-39	11011001	-71	10111001	-103	10011001
-8	11111000	-40	11011000	-72	10111000	-104	10011000
-9	11110111	-41	11010111	-73	10110111	-105	10010111
-10	11110110	-42	11010110	-74	10110110	-106	10010110
-11	11110101	-43	11010101	-75	10110101	-107	10010101
-12	11110100	-44	11010100	-76	10110100	-108	10010100
-13	11110011	-45	11010011	-77	10110011	-109	10010011
-14	11110010	-46	11010010	-78	10110010	-110	10010010
-15	11110001	-47	11010001	-79	10110001	-111	10010001
-16	11110000	-48	11010000	-80	10110000	-112	10010000
-17	11101111	-49	11001111	-81	10101111	-113	10001111
-18	11101110	-50	11001110	-82	10101110	-114	10001110
-19	11101101	-51	11001101	-83	10101101	-115	10001101
-20	11101100	-52	11001100	-84	10101100	-116	10001100
-21	11101011	-53	11001011	-85	10101011	-117	10001011
-22	11101010	-54	11001010	-86	10101010	-118	10001010
-23	11101001	-55	11001001	-87	10101001	-119	10001001
-24	11101000	-56	11001000	-88	10101000	-120	10001000
-25	11100111	-57	11000111	-89	10100111	-121	10000111
-26	11100110	-58	11000110	-90	10100110	-122	10000110
-27	11100101	-59	11000101	-91	10100101	-123	10000101
-28	11100100	-60	11000100	-92	10100100	-124	10000100
-29	11100011	-61	11000011	-93	10100011	-125	10000011
-30	11100010	-62	11000010	-94	10100010	-126	10000010
-31	11100001	-63	11000001	-95	10100001	-127	10000001
-32	11100000	-64	11000000	-96	10100000	-128	10000000

2. Look at the most significant bit (leftmost bit) of the number. If this "sign" bit is a 0, the 16 bits represent a positive number from 0000000000000000 (0) through 0111111111111111 (32,767). Convert to decimal by CFBD. If the sign bit is a 1, the 16 bits represent a negative number from 1111111111111111 (-1) through 1000000000000000 (-32768). In this case, follow step 3.

3. For a negative number: Take the binary value and change all 0s to 1s and all 1s to 0s, as shown in Figure TCNU-1.

4. Now add 1 to the result. Add 0 and 0 to get 0, 0 and 1 to get 1, 1 and 1 to get 0 and a carry. Add in any carry to the next "bit position;" this may produce another carry.

5. The result is a positive number which can be converted to decimal by CFBD. After the conversion, tack on a minus sign on the front, and you have the negative number represented. One catch: 1000000000000000 is -32,768 and produces 1000000000000000 again!

To convert a negative decimal number to two's complement form, convert the absolute (positive) value to binary by CFDB, and then follow steps 3 and 4 above, as shown in Figure TCNU-2. The result is the two's complement form of the original number.

If all of this gives you a headache, you're not alone. Working with two's complement numbers is tedious, to say the least. Maybe some day all of this will be computerized!

Figure TCNU-1 – *Two' Complement Conversion*

ORIGINAL NUMBER = 1Ø11Ø1Ø1

<u>STEP 1: EXAMINE SIGN BIT</u>

1Ø11Ø1Ø1

"SIGN" BIT IS A 1, THEREFORE NUMBER IS NEGATIVE AND MUST BE CONVERTED BY TWO'S COMPLEMENT METHOD

<u>STEP 2: CHANGE ALL Ø'S TO 1'S AND ALL 1'S TO Ø'S</u>

1Ø11Ø1Ø1

Ø1ØØ1Ø1Ø

<u>STEP 3: ADD 1</u>

```
Ø1ØØ1Ø1Ø
      +1
Ø1ØØ1Ø11
```

<u>STEP 4: CONVERT TO DECIMAL BY XXXX</u>

Ø1ØØ1Ø11 = $75_{1Ø}$

<u>STEP 5: ADD MINUS SIGN</u>

Ø1ØØ1Ø11 = $75_{1Ø}$; $-75_{1Ø}$ = ORIGINAL NUMBER

Figure TCNU-2 – *Converting from Negative Decimal*

ORIGINAL NUMBER = $-2ØØØ_{1Ø}$

<u>STEP 1: CONVERT ABSOLUTE VALUE TO BINARY BY CFDB</u>

ABS($-2ØØØ_{1Ø}$) = $2ØØØ_{1Ø}$ = $\underbrace{\text{ØØØØØ11111Ø1ØØØØ}}_{\text{16-BIT EQUIVALENT}}$

<u>STEP 2: CHANGE ALL Ø'S TO 1'S AND ALL 1'S TO Ø'S</u>

ØØØØØ11111Ø1ØØØØ

11111ØØØØØ1Ø1111

<u>STEP 3: ADD 1</u>

```
11111ØØØØØ1Ø1111
              +1
11111ØØØØØ11ØØØØ = BINARY FORM
                   OF ORIGINAL #
```

TGBB
TRSDOS, Getting Back to From BASIC, Model II

SYSTEM.

THT3
TRSDOS, How to Load, All Systems

1. Turn on computer (see TOCH).

2. Insert TRSDOS diskette (DINS).

3. Press RESET (RBWI) for Models I/III and Color Computer. (The Model II does not require a RESET, but you can use it at any time to restart the load).

4. The screen should clear, and after about 5 seconds you should see a title message. The message will be similar to this one for the Model III:

TRS-80 Model III TRSDOS version 1.3 Fri May 1, 1981

48K System, Number of Drives=2 Serial #:11240510000055

(c) (p) 1980 TANDY CORPORATION. All rights reserved

Unauthorized reproduction of this software is prohibited and is in violation of United States copyright laws

```
Enter Date (MM/DD/YY)?
```

5. Enter the date by typing in 05/23/82 followed by ENTER or similar date. You must have two digits for month, day and year, except for the Model II, which requires four year digits.

6. You should now see

```
Enter Time (HH:MM:SS)?
```

You may enter time such as 06:41:42 followed by ENTER or similar time or simply press ENTER (use periods to separate the HH.MM.SS on the Model II). If the time is not entered, the system will assume a time of 00:00:00 (midnight) for its "real time clock."

7. The PROMPT

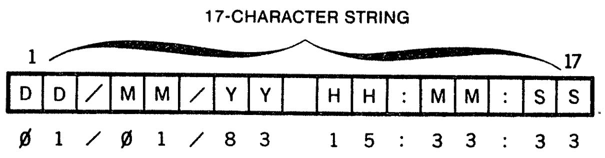
```
TRSDOS READY
```

should now be displayed. You are in TRSDOS ready to load BASIC or other programs, or run TRSDOS functions.

TIBP
Time, in BASIC Program, Most Systems

Model I, Disk BASIC, Model III, Level III or Disk only: The TIME$ function gets the time as a string in the format DD/MM/YY HH:MM:SS as shown in Figure TIBP-1.

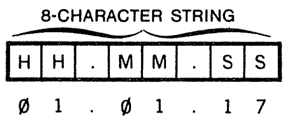
Figure TIBP-1 – *TIME$ Function Format (non-Model II)*


17-CHARACTER STRING
1
17
D D / M M / Y Y H H : M M : S S
Ø 1 / Ø 1 / 8 3 1 5 : 3 3 : 3 3

Model II only: The TIME$ function gets the time in the format HH.MM.SS, as shown in Figure TIBP-2.

For all models, of course, the current time must be valid; this means that the time must have been entered by the appropriate command. See TSRT, CDHT, RTCN.

Figure TIBP-2 – *TIME$ Function Format (Model II)*



Color Computer Extended Color and Disk BASIC: The TIMER function allows you to set a real time clock that counts in 60ths of a second up to 65,535 counts (1092 seconds or 18.2 minutes). Set the time to 0 or another value by

```
1ØØ TIMER=Ø
```

You can then find the elapsed time in seconds by

```
2ØØ PRINT INT(TIMER/6Ø)
```

Sorry, what do you want for $500?

TM13
Tokens in BASIC, Model I/III

Model I/III BASIC uses a special form of encoding for BASIC commands. BASIC commands such as LIST, NEW, FOR, RETURN and all others are not represented by their ASCII (see ADFW) equivalent text, but as one-byte codes of 128 through 250. This decreases the amount of memory storage required for BASIC programs. Tokens are always used in in-memory storage of BASIC programs, but BASIC programs may optionally be stored on disk as ASCII files (see AFWA), with "straight text" and no tokens.

The token codes used in Model I/II/III BASIC are shown in Table TM13-1 below.

Table TM13-1 – *Model I/III BASIC Token Codes*

Value	Command	Value	Command	Value	Command	Value	Command
128	END	161	ON	194	ERL	227	TAN
129	FOR	162	OPEN	195	ERR	228	ATN
130	RESET	163	FIELD	196	STRING$	229	PEEK
131	SET	164	GET	197	INSTR	230	CVI
132	CLS	165	PUT	198	POINT	231	CVS
133	CMD	166	CLOSE	199	TIME$	232	CVD
134	RANDOM	167	LOAD	200	MEM	233	EOF
135	NEXT	168	MERGE	201	INKEY$	234	LOC
136	DATA	169	NAME	202	THEN	235	LOF
137	INPUT	170	KILL	203	NOT	236	MKI$
138	DIM	171	LSET	204	STEP	237	MKS$
139	READ	172	RSET	205	+	238	MKD$
140	LET	173	SAVE	206	-	239	CINT
141	GOTO	174	SYSTEM	207	*	240	CSNG
142	RUN	175	LPRINT	208	/	241	CDBL
143	IF	176	DEF	209	[	242	FIX
144	RESTORE	177	POKE	210	AND	243	LEN
145	GOSUB	178	PRINT	211	OR	244	STR$
146	RETURN	179	CONT	212	>	245	VAL
147	REM	180	LIST	213	=	246	ASC
148	STOP	181	LLIST	214	<	247	CHR$
149	ELSE	182	DELETE	215	SGN	248	LEFT$
150	TRON	183	AUTO	216	INT	249	RIGHT$
151	TROFF	184	CLEAR	217	ABS	250	MID$
152	DEFSTR	185	CLOAD	218	FRE		
153	DEFINT	186	CSAVE	219	INP		
154	DEFSNG	187	NEW	220	POS		
155	DEFDBL	188	TAB(	221	SQR		
156	LINE	189	TO	222	RND		
157	EDIT	190	FN	223	LOG		
158	ERROR	191	USING	224	EXP		
159	RESUME	192	VARPTR	225	COS		
160	OUT	193	USR	226	SIN		

TMER
TM Error

Type Mismatch error.

Typically, you have used a numeric variable name where you should have used a string variable name, or vice versa, as in

```
100 A$=LEFT$(B,3)   'B should be B$
```

TMTW
Tokens in BASIC, Model II

Model II BASIC uses a special form of encoding for BASIC commands. BASIC commands such as LIST, NEW, FOR, RETURN and all others are not represented by their ASCII (see ADFW) equivalent text, but as one-byte codes of 128 through 254. This decreases the amount of memory storage required for BASIC programs. Tokens are always used in in-memory storage of BASIC programs, but BASIC programs may optionally be stored on disk as ASCII files (see AFWA), with "straight text" and no tokens.

To see token compression, LIST a BASIC file and look at the ASCII hexadecimal representation.

The token codes used in Model II BASIC are shown in Table TMTW-1 below.

Table TMTW-1 – *Model II BASIC Token Codes*

Value	Command	Value	Command	Value	Command	Value	Command
128	END	160	OPEN	193	STRING$	229	TAB
129	FOR	161	FIELD	194	INSTR	230	ATN
130	CLS	162	GET	195	TIME$	231	CVI
131	RANDOM	163	PUT	196	DATE$	232	CVS
132	NEXT	164	CLOSE	197	MEM	233	CVD
133	DATA	165	LOAD	198	INKEY$	234	EOT
134	INPUT	166	MERGE	199	THEN	235	LOC
135	DIM	167	CLK	200	NOT	236	LOF
136	READ	168	KILL	201	STEP	237	MKI
137	LET	169	LSET	207	AND	238	MKS$
138	GOTO	170	RSET	208	OR	239	MKD$
139	RUN	171	SAVE	209	XOR	240	CINT
140	IF	172	SYSTEM	210	EQV	241	CSNG
141	RESTORE	173	LPRINT	211	IMP	242	CDBL
142	GOSUB	174	DEF	212	MOD	243	FIX
143	RETURN	175	PRINT	213	SPC	244	LEN
144	REM	176	CONT	214	TAB	245	OCT
145	STOP	177	LIST	215	FILES	246	HEX
147	TRON	178	LLIST	216	VERIFY	247	STR$
148	TROFF	179	DELETE	217	SGN	248	VAL
149	SWAP	180	AUTO	218	INT	249	ASC
150	ERASE	181	CLEAR	219	ABS	250	CHR$
151	DEFSTR	182	RENUM	220	FRE	251	SPACE$
152	DEFINT	183	NEW	221	ROW	252	LEFT$
153	DEFSNG	185	TO	222	POS	253	RIGHT$
154	DEFDBL	186	FN	223	SQR	254	MID$
155	LINE	188	USING	224	RND		
156	EDIT	189	VARPTR	225	LOG		
157	ERROR	190	USR	226	EXP		
158	RESUME	191	ERL	227	COS		
159	ON	192	ERR	228	SGN		

TOCH
Turning on the Computer, How To

For all systems: Do not have a diskette in any disk drive before powering up. The diskette contents may be destroyed, especially in the Model I.

Model I: Turn on the CPU unit by pressing in the power switch button, located on the rear of the unit just left of the three plugs (see Figure TOCH-1). Turn on the expansion interface by pressing in the clear button on the front of the unit (see Figure TOCH-2). Turn on any disk drives by setting the rear toggle switch to up (see Figure TOCH-3). Turn on any peripheral equipment such as printers by various means, usually a rear-mounted power switch on Radio Shack equipment.

Figure TOCH-1 – *Model I Power Switch*

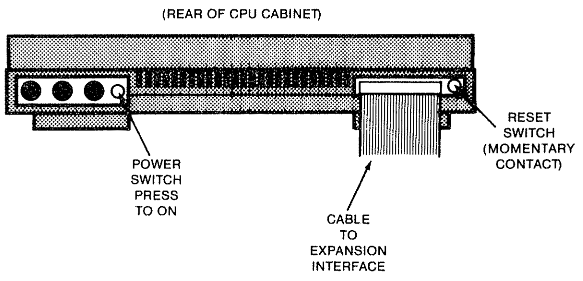

(REAR OF CPU CABINET)
POWER SWITCH PRESS TO ON
CABLE TO EXPANSION INTERFACE
RESET SWITCH (MOMENTARY CONTACT)

Figure TOCH-2 – *Model I Expansion Interface Power*

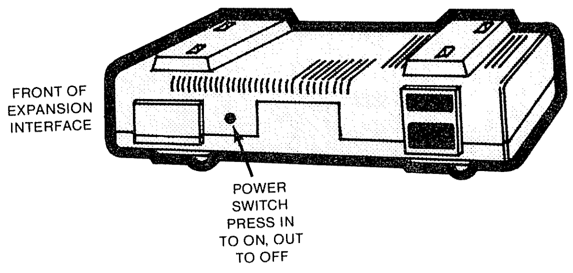
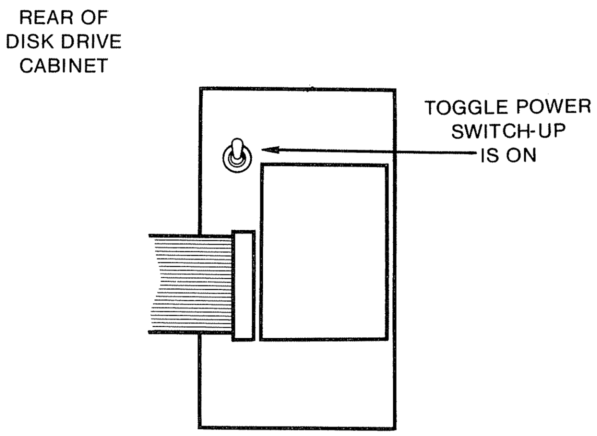

FRONT OF EXPANSION INTERFACE
POWER SWITCH PRESS IN TO ON, OUT TO OFF

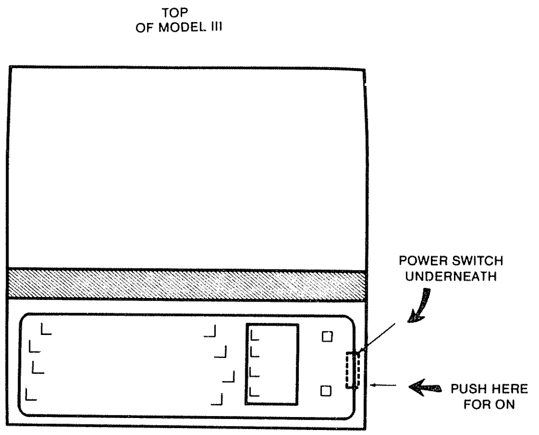
Figure TOCH-3 – *Model I Disk Drive Power*



Model II: Set POWER switch on front panel to up (ON). You should see an "INSERT DISKETTE" message displayed on the screen. Insert a TRSDOS diskette (see DINS). The Model II will automatically load TRSDOS.

Model III: Power switch is under right-hand edge of cabinet (see Figure TOCH-4). Press forward portion to turn on. Turn on peripheral equipment as under Model I.

Figure TOCH-4 – *Model III Power Switch*



Color Computer: Press in button power switch, located on left rear of cabinet. Turn on television set. Turn on disk drives and peripheral equipment as in Model I.

After powering up on the Models I/III and Color Computer, you may see "garbage" on the screen. Refer to RWBI to "reset" the system to ROM BASIC and to TRSDOS/LDOS loading procedures to load the operating system from disk.

TRHT
Tracing BASIC Programs

Tracing is available on every system except the Model I/III, Level I and Color Computer with Color BASIC.

Tracing will let you see exactly which BASIC lines are being executed so that you can see a "trace" of the BASIC program flow.

To start tracing, enter TRON. This turns on the trace capability. To stop tracing, enter TROFF. You guessed it — TROFF stops the trace.

Now, an important point: TRON and TROFF can be executed "dynamically," a fancy way of saying that these commands can be included in BASIC programs. You can therefore turn the trace on and off within the program.

A typical trace appears as in Figure TRHT-1 — gobs of line numbers spewed out so rapidly on the screen that you can't see what's happening. Try STOP and some good "desk checking" in lieu of this method of debugging — it's overrated.

If you must use tracing, data generated by PRINT or other commands will be interspersed with the trace data. Pressing SHIFT and the "@" key simultaneously will stop the display; pressing any key thereafter will restart.

Figure TRHT-1 – *Typical Trace in BASIC*

```
<1ØØ><11Ø><111><115><12Ø><11Ø><111><115><12Ø><11Ø><111><115>
<12Ø><11Ø><111><115><12Ø><11Ø><111><115><12Ø><11Ø>MIDPOINT
<111><115><12Ø><11Ø><111><115><12Ø><11Ø><111><115><12Ø><11Ø>
<111><115><12Ø><11Ø><111><115><12Ø><11Ø><111><115><12Ø><13Ø>
<1ØØ><11Ø><111><115><12Ø><11Ø><111><115><12Ø><11Ø><111><115>
<12Ø><11Ø><111><115><12Ø><11Ø><111><115><12Ø><11Ø>MIDPOINT
<111><115><12Ø><11Ø><111><115><12Ø><11Ø><111><115><12Ø><11Ø>
<111><115><12Ø><11Ø><111><115><12Ø><11Ø><111><115><12Ø><13Ø>
<1ØØ><11Ø><111><115><12Ø><11Ø><111><115><12Ø><11Ø><111><115>
<12Ø><11Ø><111><115><12Ø><11Ø><111><115><12Ø><11Ø>MIDPOINT
<111><115><12Ø><11Ø><111><115><12Ø><11Ø><111><115><12Ø><11Ø>
<111><115><12Ø><11Ø><111><115><12Ø><11Ø><111><115><12Ø><13Ø>
<1ØØ><11Ø><111><115><12Ø><11Ø><111><115><12Ø><11Ø><111><115>
<12Ø><11Ø><111><115><12Ø><11Ø><111><115><12Ø><11Ø>MIDPOINT
<111><115><12Ø><11Ø><111><115><12Ø><11Ø><111><115><12Ø><11Ø>
<111><115><12Ø><11Ø><111><115><12Ø><11Ø><111><115><12Ø><13Ø>
<1ØØ><11Ø><111><115><12Ø><11Ø><111><115><12Ø><11Ø><111><115>
<12Ø><11Ø><111><115><12Ø><11Ø><111><115><12Ø><11Ø>MIDPOINT
<111><115><12Ø><11Ø><111><115><12Ø><11Ø><111><115><12Ø><11Ø
<111><115><12Ø><11Ø><111><115><12Ø><11Ø><111><115><12Ø><13Ø>
<1ØØ><11Ø><111><115><12Ø><11Ø><111><115><12Ø><11Ø><111><115>
<12Ø><11Ø><111><115><12Ø><11Ø><111><115><12Ø><11Ø>MIDPOINT
<111><115><12Ø><11Ø><111><115><12Ø><11Ø><111><115><12Ø>
Break in 1ØØ
READY
>
```

TRUS
TRACE, Using, Model I TRSDOS, Model I/III LDOS

A weird command, this one. Although the original Model I manual states that it can be "invaluable" in debugging machine language programs, I suspect it is seldom used.

```
TRACE (ON) or TRACE (OFF)
```

will turn the trace on or off. When the trace is on, you'll see the contents of the Z-80 Program Counter register in the upper right-hand corner of the screen. This is not a continuous representation of the PC, but is updated hundreds of times per second.

Marginally useful in seeing where your assembly language program is executing, but you've got DEBUG (DT1U) for that.

TSAL
Table Size in Assembly Language

Why not let the assembler do the work for you? Let it calculate the size of a table or other data structure.

In Z-80 code:

```
TABLE   EQU    $        ;start of table
        DEFB   12       ;data
        .
        .
        DEFB   37       ;end of data
TABS    EQU    $-TABLE  ;use TABS in LD  IX,TABS
```

In 6809 code:

```
TABLE   EQU    *        start of table
        FCB    12       data
        .
        .
        FCB    37       end of data
TABS    EQU    *-TABLE  use TABS in LDX  #TABS
```

TSPD
To Stop the Display in BASIC, All Systems

If your BASIC program is tracing or listing rapidly and you can't see the display, stop it by pressing the SHIFT and "@" keys simultaneously on the Model I, III, and Color Computer. On the Model II, press the "HOLD" key. Continue by pressing any key. Repeat whenever necessary.

TSRT
TIME, Setting the Real-Time-Clock, TRSDOS/LDOS, Model I/II/III

If you want to use the real-time-clock in your system as an accurate chronometer, read RTCN.

The time can be set on startup or reset at any time by

```
TIME hh:mm:ss   (Model I/III)
TIME hh.mm.ss   (Model II)
```

where hh, mm, and ss are 2-digit values for hours, minutes and seconds, respectively. Use leading zeros, as in

TIME 23:12:13

Notice that TRSDOS/LDOS uses a "24-hour" clock. You ex-military people will know about this, or even those of you who have heard John Wayne say "we'll make the bombing run at fifteen-hundred hours." In the 24-hour clock, 1 pm becomes 13 hours, 2 pm 14 hours and so forth, up to 11:59:59 p.m., which becomes 23:59:59. Just add 12 hours to p.m. times, and y ı'll be able to rendezvous with the Big Duke.

TSTP
To Stop the Program

Hit the BREAK key, bozo!

Oh, oh! Didn't work, eh? Are you waiting for a cassette tape to read, and are you at a point where there is no data on the tape? No, eh? . . . . Is the printer on-line? You're not printing . . . did you call a machine language program that has not been fully debugged? You aren't using machine language code . . . do you have a lot of string manipulations in your program? If so, the BASIC interpreter may be "cleaning up" the string storage area and may get back to you in several minutes or even 15 minutes or more. No, eh?. . . . Did the program try to use a disk file and you had the diskette out? That's not it, eh? . . . .

If you've eliminated all of the above, you may have a hardware problem. Resetting the system (see RBWI) will save your BASIC program, but not the results, unless you do a GOTO from where you stopped (see BPIB). Best to save your program on cassette or disk if you don't already have a copy and try again.

UEER
UE Error

Unprintable ($#"#!!) error. My favorite. The BASIC interpreter, it seems, is not above a little foul language. (This is now called "Undefined Error," a name that is not nearly as much fun).

Actually, you've tried to generate an error by using an invalid error code in an ERROR statement. See ETIB.

ULER
UL Error

Undefined line error. You've done a GOTO or GOSUB to a non-existent line in your program.

UTFB
Using Trigonometric Functions in BASIC

(Does not apply to Model I/III, Level I and Color Computer BASIC). First of all, let's make the assumption that if you *are* using trigonometric functions that you know something about trigonometry, analytic geometry, or some other math science that uses SIN, COS, TAN, and the like. If not, you're not going to get it here (and we'll darn well not give you your money back for the book either . . .).

There are three basic trigonometric functions implemented in BASIC: SIN, COS, and TAN, and one inverse trigonometric function, ATN. The argument for the first three is in radians, and the result of the fourth will be in radians. In case you're rusty, there are 2*pi radians in 360 degrees; simply divide the argument in degrees by 57.29578 or multiply the result by the same value to get degrees. As in all functions, enclose the argument in parentheses.

Use other common trigonometric identities to derive other trigonometric functions. See SRBH for a sample BASIC program using SIN.

notes

VAIF
Visible and Invisible Files, What Are They?

An invisible file is one that does not show up on a simple DIR listing. A visible file will be displayed or printed. That's it — not too profound. Of course, you can still get a DIR listing that will display *all* files, visible and invisible. The reason for invisible files? Suppose that you always have a dozen BASIC programs on your diskette that are involved with your business programs. It's nice to make them invisible so that you don't have them listed on the DIR listing, obscuring the data files you're interested in seeing.

Initially, all files you create are visible, but you can make them invisible by using the ATTRIB command (ADFC, ADFL).

VMHT
Voltmeter, How to Use

A typical voltmeter is shown in Figure VMHT-1. If you're going to buy a voltmeter, get a "high-impedance" digital one instead of the swinging needle version. In most cases the digital VM will be best. (The swinging needle of an "analog" VM, however, is handy for detecting "blips" which will cause the digital VM to spew out meaningless digits. The digital VM does take some time to settle down to a "reading" — about 3/4 second).

Voltmeters measure direct-current voltage, alternating-current voltage, resistance and dc or ac current. Most of the time you'll be measuring voltage or resistance. Most digital voltmeters are "overload" protected (fused) and can be used by novices with impunity.

Using the voltmeter is simplicity in itself. Set the selection switch to measure DC V, AC V, KOHM, DC ma (milliamps) or AC ma. Then set the range knob for the lowest scale on which you get a valid reading. Invalid readings result in a constant "1 . " or similar indication. Short circuits are close to 0 ohms. High resistance (no conductivity) is "1 . " or similar. When measuring currents, break the circuit at the point at which current is flowing and use the DVM leads to complete the circuit (see Figure VMHT-2).

Figure VMHT-1 – *Typical Digital Voltmeter*

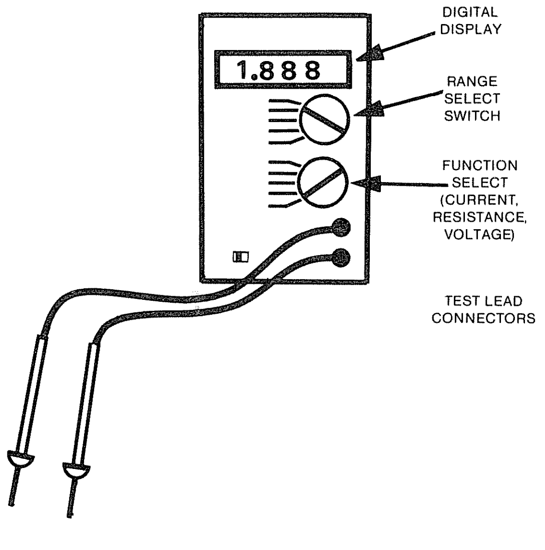
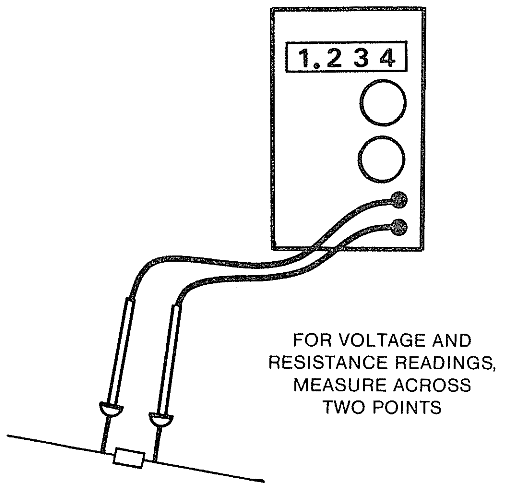
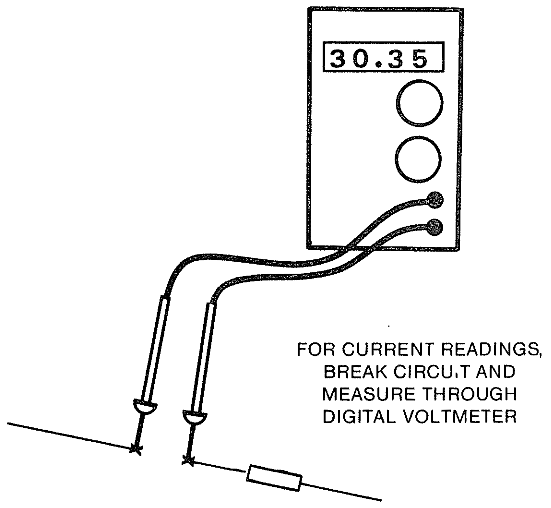


Figure VMHT-2 – *Current Measurements Using DVM*


1.234
FOR VOLTAGE AND RESISTANCE READINGS, MEASURE ACROSS TWO POINTS


30.35
FOR CURRENT READINGS, BREAK CIRCUIT AND MEASURE THROUGH DIGITAL VOLTMETER

You can use a high-impedance DVM to read any logic circuit values, as long as they are "steady-state." Use a logic probe (see LPHT) or oscilloscope to test rapidly changing signals.

The current RS catalog carries three, from about $60 to $90 — a worthwhile investment in one of the most usable computer/electronics tools.

VTTO

VERIFY, to Turn On and Off, Model I TRSDOS, Model I/III LDOS, Color DOS

Use

```
VERIFY (ON)    (Model I/III)
VERIFY (OFF)   (Model I/III)
```

or

```
VERIFY ON    (Color Computer)
VERIFY OFF   (Color Computer)
```

VTTT

VERIFY, to Turn On and Off, Model II TRSDOS

Use

```
VERIFY ON
VERIFY OFF
```

Entering

```
VERIFY DETECT=ON
```

or

```
VERIFY DETECT=OFF
```

will turn the identification sense feature on or off. If the identification sense feature is on, TRSDOS will automatically check the diskette identification before any disk read or write. This ensures that a new diskette has not been swapped with the original. If the id does not match, no I/O to the disk can occur.

VWDI

VERIFY - What Does It Do?

A verify operation on disk writes in the ideal case writes a disk file, sector by sector (see DSHL) and then reads back in the disk file, comparing every byte of data to make certain that the file has been written properly. In the practical case on most of the TRS-80s the byte-by-byte sector compare is really a write followed by a read with a comparison of the "CRC," a checksum from the disk controller logic. The effect is almost as good as a byte-by-byte compare and is a good check on the validity of the data.

The verify operation can be optionally turned on in most DOSes, including TRSDOS and LDOS (see VTTO and VTTT). It is normally on for FORMAT and BACKUP operations (see BDSM) and for updating the directory on diskettes (DIDI).

Because the verify involves additional "overhead," typically another disk read after write, it does increase the overall time to write a file — probably by about 50%. Is it worth it for normal user writes? The disk is reliable enough so that the answer is no, in most applications. Rely instead upon a good backup philosophy (BDSM).

WCCD
"Wild-Card" Copies of Disk Files, Model III TRSDOS

The COPY format

```
COPY /ext:n :m
```

where /ext is the file name extension (see FNMH), :n is the source disk drive and :m is the destination disk drive, will cause a copy of all disk files with the specified extension from drive n to m. As an example, suppose you had a number of files on drive 1 that had the extension "/BAS," for BASIC.

```
COPY /BAS:1 :Ø
```

would cause a copy of all the /BAS files onto the diskette in drive 0.

This COPY is a convenient way to copy similar files "en masse."

WHHT
Wire-Wrapping, How to

Wire-wrapping is a proven way of rapidly connecting integrated circuit sockets. "Rapidly" is subjective. Expect from one to two wraps per minute.

1. Go to your local Radio Shack store and pick up the following equipment:

   A. Wire-wrapping tool (276-1570).

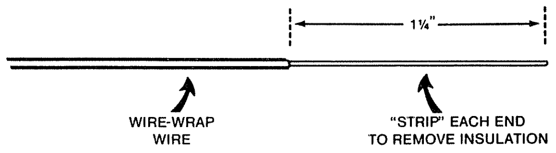
   B. Roll of 30-gauge wire-wrap wire, any color (278-501).

   C. Wire stripper (one is included with this tool), or use RS 64-2129.

2. Cut a length of wire for the connection. Cut about 2 1/2 inches more than you'll need to go from one connection to the other. Length of the wire is not critical as it is in some radio or television equipment.

3. Strip the ends of the wire as shown in Figure WWHT-1.

Figure WWHT-1 – *Stripping Wire-Wrap Wire*



4. Insert one end of the wire into the wire-wrapping tool as shown in Figure WWHT-2.

5. Place the tip of the wrap tool over the wire-wrap pin and down to the board on which the pin is mounted as shown in Figure WWHT-3.

6. Rotate the tool clockwise about 6 complete turns. Don't press down on the tool with too much pressure; on the other hand, apply *some* pressure. (Maybe what we need here is a micro "torque wrench"). Figure WWHT-4 shows a correct and incorrect wrap.

7. Repeat steps 4 through 6 for the other end of the wire.

There are two general types of integrated circuit sockets: one with short pins (about 3/8 inch) and one with longer pins (about 5/8 inch). You should have no problem fitting three connections on either type. Generally, you won't need more than one or two connections on any pin, unless you're building a large-scale project.

Figure WHHT-2 – *Wire Wrap Tool Insertion*

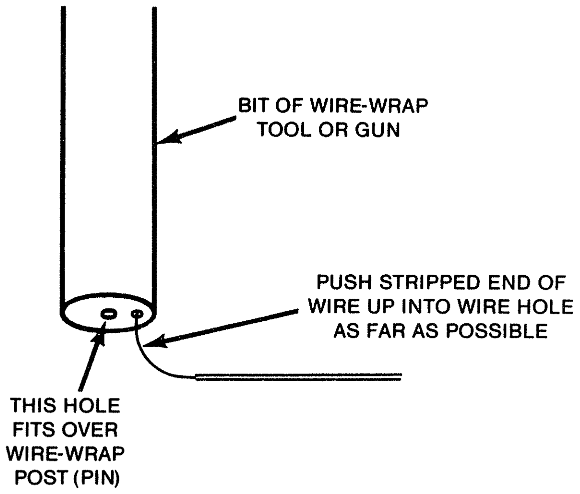

BIT OF WIRE-WRAP TOOL OR GUN
PUSH STRIPPED END OF WIRE UP INTO WIRE HOLE AS FAR AS POSSIBLE
THIS HOLE FITS OVER WIRE-WRAP POST (PIN)

Figure WHHT-3 – *Wire Wrap Tool Position*

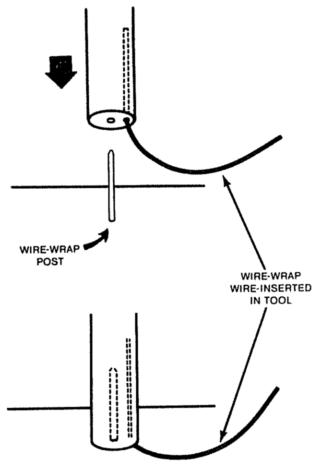
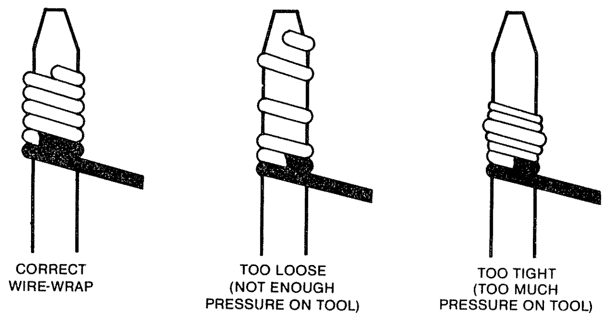


Figure WHHT-4 – *Wire Wraps*


CORRECT WIRE-WRAP
TOO LOOSE (NOT ENOUGH PRESSURE ON TOOL)
TOO TIGHT (TOO MUCH PRESSURE ON TOOL)

Z8AF
Z-80 Address Format

Confused about storing USR call addresses or getting string addresses? It isn't bad enough that you have to convert between decimal and hexadecimal, but then the darn things are reversed! Sorry, that's the way the Z-80 microprocessor in the Model I, II,and III works. It looks for an address value least significant byte followed by most significant byte. If you had these values in two memory locations:

```
1ØØ,128
```

then the memory address represented would be 128 (most significant byte), 100 (least signicant byte), or 128*256+100, or 33,024 decimal (see CFHD). *All two-byte memory addresses are arranged in that fashion*, and you'll have to store them this way for USR calls (see RCWA) and other machine-language functions.

ZUEC
ZBUG, Using, EDTASM+ , Color computer

This discussion requires that you know something about 6809E assembly language and the EDTASM+ assembly process. Read MLWI and ALWI if you know nothing about these operations. You also need to know something about hexadecimal operations. Read CFDH. After you read these procedures come back here.

Aha, I knew you'd be back . . . let's see what transpired in your absence . . . War in the Aleutians, IBM bought by Radio Shack . . .

ZBUG, General Description: If you haven't used ZBUG, you're in for two surprises. One — it's somewhat complicated. Two — it's very powerful. ZBUG is a debug package that you can use in conjunction with in-memory assembly. It is set up to reference object code by symbolic locations, derived from the assembler symbol table. It can work in octal, decimal, hexadecimal; it can display data in symbolic (**label** START, for example) or numeric fashion, one or two bytes at a time. All hype aside, in-memory assembly and ZBUG are about as close to heaven as 6809E assembly-language programmers can get — well, maybe a macro capability . . . .

ZBUG, Getting To and From: From the EDTASM+ command mode, type Z. You'll see a # prompt indicating that you're in ZBUG. The # prompt is the ZBUG command mode and you should be in this mode to to activate the ZBUG commands. To get back into ZBUG command mode, press BREAK. To get back to the Editor, enter E, and you'll return to the * prompt of the Editor.

Examining Memory in Hexadecimal Format: To examine any memory location, type the memory location, followed by a slash. On entry to ZBUG, you'll see something like

```
#FF/     ROR   <Ø
```

What you're seeing is the contents of memory location **$00FF** in "symbolic" form. If you first type B, you'll see the "byte mode" form, which is more recognizable:

```
#B
#FF/     6
```

To examine consecutive locations, press the down arrow. To back up, use the up arrow. Typical display

```
1Ø6/     7E     (down arrow)
1Ø7/     ØE2    (down arrow
1Ø6/     7E     (up arrow)
```

To Examine Memory in ASCII Format: While in ZBUG command mode, enter A. You can now examine locations with the slash and have the contents printed in ASCII. The up arrow and down arrow "scrolls" up and down as before:

```
#A
114/     J      (down arrow)
115/            (down arrow)
116/     O
```

Note that invalid ASCII characters are not printed.

To Examine Memory in Mnemonic Mode (Disassembly): While in ZBUG command mode, enter M. You can now examine locations with the slash and have the contents printed in 6809E instruction mnemonic form, or "disassembled." Typical display:

```
#M
114/     DECA            (down arrow)
115/     SUBA    #4F     (down arrow)
117/     ??
```

Note that the locations increment according to the length of the instruction. Some data does not represent valid 6809E instructions, therefore EDTASM+ prints ??.

To Examine Memory Two Bytes at a Time (Word Mode): Enter W while in ZBUG command mode. You can now examine locations in "word" mode. In this mode, ZBUG assumes the data is two bytes long and prints two consecutive locations. This is handy for displaying addresses. Example:

```
#W
1ØB/     7E       (down arrow)
1ØD/     894C     (down arrow)
1ØF/     7EAØ
```

Note that once you've started from a location, ZBUG increments two bytes at a time, therefore "sync up" to the proper byte location initially.

To Examine Your Program in Memory: Run an in-memory assembly (see EAIM). All of the symbols will be put into an assembler symbol table, a standard way of translating from source to object code. They will also be available for use by ZBUG, providing that you've chosen the proper "display mode." Whenever you examine a memory location and that memory location is within your program, ZBUG will print out the address in symbolic form.

Suppose that we have a short program as follows:

```
START    LDA   $12     load that barge
         LDX   START   start the end
ENDLOC   LDA   #$12    th..th..th..that's all folks!
         END
```

If we do an in-memory assembly, we'll wind up with a symbol table of

```
ENDLOC   ØØ88A
START    Ø885
```

(Your assembly may differ).

If we now go to ZBUG and enter START, we'll see:

```
#START/  LDA  <12
 START+2 /                LDX  >START
 ENDLOC/                  LDA  #12
```

ZBUG used assembly symbolic references both for input and output.

To get back to numeric "display mode", enter N on entry to ZBUG. ZBUG will then ignore any symbols from in-memory assembly:

```
#N
885/    LDA  <12     (down arrow)
887/    LDX  >885    (down arrow)
88A/    LDA  #12     (BREAK)
#B                   (set Byte examination mode)
885/    96
```

To display your program location values as symbols and the contents as numeric, set the "half symbolic" display mode by entering H:

```
#H
885/     96
START+1/         12
START+2 /        ØBE
```

To Examine Memory in Decimal or Octal: The "default" display (output) base is hexadecimal, or 16 (see CFDH). You can change this to decimal or octal by the O command:

```
#1ØØØ/     ØCE     (down arrow)
 1ØØ1/     ØFF     (BREAK)
#O1Ø               (set decimal)
#1ØØØ/     2Ø6T    (down arrow)
 4Ø97T/    255T    (down arrow)
 4Ø98T/    255T    (BREAK)
#O8                (set octal)
#1ØØØ/     316Q    (down arrow)
 1ØØØ1Q/        377Q
 1ØØØ2Q/        377Q
```

Note that values with a T suffix are decimal, values with a Q suffix are octal, and values without any suffix are hexadecimal. Note also that changing the output mode base did not affect the input mode base. The input of 1000 was still interpreted as 1000 hexadecimal.

Examining Memory Locations and Changing the Contents: To examine memory and change the contents, examine by the procedures above and before pressing down (or up) arrow, type the new value. Suppose you wanted to change locations $1000, $1001, and $1002 to $12, 102T, and 77Q, respectively. You could do something like this:

```
#1ØØØ/    33     12     (down arrow)
 1ØØ1/    1      1Ø2T   (down arrow)
 1ØØ2/    1      77O    (down arrow)
 1ØØ3/    1             (up arrow)
 1ØØ2/    3F            (up arrow)
 1ØØ1/    66            (up arrow)
 1ØØØ/    12
```

Note that the input values used a suffix to determine the "base" - 16, 10, or 8. Note also that the output mode still remained in the default of hexadecimal when the locations were redisplayed.

Changing the Input Mode Base: The input mode base default is hexadecimal, base 16. To change to decimal or octal, enter I10 or I16. After you do this, all input data will be assumed to be that base without a suffix of T or Q. To change back to hexadecimal, enter I16. You can always use the H (exadecimal) T or Q suffix to "force" another input base for a single value. Example:

```
#I1Ø                  (set decimal input mode)
2ØØØ/   ØC            (still hex output)
7D1/    ØE2           (7D1 hex is 2ØØ1 decimal)
7D2/    3Ø     18H    (change to 34 decimal)
7D3/    44     1ØQ    (change to 8 decimal)
```

To Display Numeric While in the Symbolic Mode: The semicolon (;) and equal sign can be used to "force" display in numeric mode and numeric/byte mode, respectively. Enter one or the other directly after the contents of a location is printed:

```
#885/    LDA    <12   =96    (force numeric/byte)
```

Z

To Display the Contents of Registers: To display the contents of all 6809E registers, enter R

```
#R
A=ØØ B=ØØ DP=ØØ CC=ØØ =
X=ØØØØ Y=ØØØØ U=ØØØØ S=Ø777
PC=ØØØØ
```

To Display and Change the Contents of A Single Register: Enter the register mnemonic, followed by a slash. You can now change the contents by entering data:

```
#A/    12      (A register)
#B/    34      (B register)
#CC/   Ø=      (condition codes)
#X/    1234    (IX)
#Y/    567A    (IY)
#S/    777     (hardware stack pointer)
#U/    1ØØØ    (user stack pointer)
#PC/   2ØØ2    (program counter)
#DP/   Ø       (DP register)
```

You can "force" the display of any address in RAM in *condition code format* by entering a colon (:) after the contents. This is handy when examining data in the stack or elsewhere that represents CC flags:

```
#1ØØØ/  12 :=IV   (equals, interrupt, overflow)
```

To Display A Block of Memory Locations: To display a block of memory in the current display and examination modes, use the T command:

```
#T1ØØØ 1Ø2Ø
```

displays memory from locations 1000 through 1020. There must be a space between the starting and ending locations.

To Print a Block of Memory Locations: Use the TH command in the same format as the T command:

```
#TH1ØØØ 1Ø2Ø
```

prints the contents of locations 1000 through 1020 on the system line printer through the RS-232-C port.

To Copy a Block of Memory Locations: To copy a block of memory from one set of locations to another, use the U command:

```
U sourceaddr destaddr count
```

To move locations **$1000** through **$1010** to locations **$2000** through **$2010**, for example, you'd have

```
U 1ØØØ 2ØØØ 11
```

Note the count here is in hexadecimal.

To Save and Load Core Image on Cassette: What in the name of von Neumann is "core"? An archaic term referring to RAM in magnetic cores — tiny, doughnut-shaped devices that stored one bit. In any event, ZBUG can easily dump any section of RAM (or ROM) on cassette tape. The resulting data can be loaded back by ZBUG (Whew! Thank goodness they included a Load command . . .) or by CLOADM in BASIC.

To save any block of memory, use the P (for Print, another anachronism)

```
#P name startaddr endaddr execaddr
```

To save locations **$1000** through **$1800** with file name "JUNQUE" and execution address **$1000**, for example, you'd enter

```
#P JUNQUE 1ØØØ 18ØØ 1ØØØ
```

To load back in a saved file, you'd just enter L with or without a file name. If no file name is specified, ZBUG assumes you know what you're doing (how wrong it is!) and will load the next file from cassette:

```
#L          (loads next file)
#L JUNQUE   (loads file JUNQUE)
```

To verify what you've recorded, use the V command. It will read back the cassette file and compare it with what is in the memory block used in the P command. If you've recorded JUNQUE properly, you can now power down with impunity:

```
#V JUNQUE   (verify JUNQUE)
```

The cassette P and L commands are indispensable in program debugging. After debugging and patching locations, you can "checkpoint" the patched program by writing it to tape. You can always recover the last program saved on tape even though your current program may explode and scatter bits and pieces from Mission Viejo to Fort Worth.

How to Breakpoint a Program: First of all read BPFM for breakpoint philosophy. To set a breakpoint at any location, use X followed by the location. You can use up to 8 breakpoints:

```
#XSTART+45      (set breakpoint at START+45 bytes)
#X2ØØØ          (set breakpoint at location $2ØØØ)
```

To display all of the current breakpoints, enter D:

```
#D
Ø BRK @ 8CA     (breakpoint Ø)
1 BRK @ 2ØØØ    (breakpoint 1)
```

To delete any breakpoint, use the Y(ank) command. If you enter Y alone, all breakpoints will be deleted. If you specify an address, only the one breakpoint will be deleted:

```
#Y              (yank all breakpoints)
#Y2ØØØ          (yank the breakpoint at loc'n 2ØØØ)
```

To Transfer Control to Any Location: Okay, you've breakpointed and are ready to run your program. Use the G command to transfer control to any location:

```
G1ØØØ
```

for example, transfers control to location **$1000**. Let's see, if I type in G1000, I should get a good display of a menu on the CoCo screen. Here I go . . . .

SELECT ONE OF THE FOLLOWING:

```
1. ENTER NAME$$###$$$>>>>>>>>,,,,,,>>>>>>>>>>,,,,,
>>>><<<<MMMUUYYYEER##$$%&%&&´(´()())*****))))()((
DSS"#!#$%&´(())****=*===*)))((((´´´&&&&%%%$%$$$$$
```

Ah well, back to the debugging. For more on ZBUG, see:

ECCE for ZBUG Expressions and Calculator Mode.
BPFM for General Debugging.

Model 100 Procedures

This special Model 100 addendum has been compiled to give you up to date information about the TRS-80 Model 100. In doing so, we've looked at the material in the "main sequence" of procedures and seen which ones apply to the Model 100 as well. Many of the procedures in the main sequence apply directly to the Model 100 – the Model 100 BASIC is Microsoft BASIC, the error codes are similar, and general subjects are applicable to all machines. For the considerable new material on the Model 100, we've compiled many new procedures.

Model 100 Index

Here's how to find a general topic: Look in the main sequence index for the topic. If the procedure doesn't exclude certain machines, it'll apply to the Model 100 as well. Procedure DLSC, "Digital Logic, A Short Course" applies to all Radio Shack computers, for example.

Here's how to find a Model 100 topic: First, look in the main sequence index under the topic. There you'll find a "keyword" of four characters. Now, go to the Model 100 section and look for that keyword. If the topic applies equally well to the Model 100, you'll find a procedure in the Model 100 section that describes the topic as used on the Model 100.

Let's say you're looking for cursor control actions. If you look in the main sequence index under "Cursor, finding where it is", you'll find the keyword CFOW. Looking in the Model 100 section, you'll find the procedure CFOW-100, which describes Model 100 cursor commands. The "-100" denotes that a main sequence procedure either applies directly to the Model 100, or describes slight differences between the Model 100 and other systems.

If you can't find the Model 100 topic in the main sequence index, then go to the Model 100 index and look for the topic there. You will probably find a Model 100 keyword that applies. It will be a 4-character keyword and it will be located in the Model 100 section. An example is "Time interrupt", which references keyword TIMO. Procedure TIMO is located in the Model 100 section and describes how to use the Model 100 "ON TIME$" interrupt.

Model 100 Index

Model 100 List of Procedures by Keyword

MWAT-100	Modems, What Are They?, Model 100
NFER-100	NF Error, Model 100
NRER-100	NR Error, Model 100
ODER-100	OD Error, Model 100
OGHU-100	ON...GOSUB, ON...GOTO, BASIC, Model 100
OMER-100	OM Error, Model 100
OSER-100	OS Error, Model 100
OVER-100	OV Error, Model 100
PHTU-100	Parentheses, How to Use, Model 100
POOM	Power Off and On, Model 100
PPKU-100	PEEk and POKE, Using, Model 100
PROT-100	Protecting Memory in BASIC, Model 100
PRUU-100	PRINT USING, BASIC, Using, Model 100
PSMO	PSET, PRESET in BASIC, Model 100
PTSC-100	Printing the Screen, Model 100
PVMO	Passing Variables to Machine Language Programs, Model 100
QWII-100	Quote, in BASIC, What Is It?, Model 100
RBWI-100	Reset Button, Where Is It?, Model 100
RCCO	RS-232-C Connector, Model 100
RCWA-100	ROM Calls, What Are They?, Model 100
RDHT-100	Random Data, How to Generate, Model 100
RERF	Renaming a RAM File, Model 100
RFDO	RAM File, Deleting, Model 100
RFMO	RAM, Finding MAX, Model 100
RGER-100	RG Error, Model 100
RHSO	RS-232-C Interface, How to Set, Model 100
RLIB-100	Replacing a Line in BASIC, Model 100
RNBL-100	Renumbering BASIC lines, Model 100
RNCH	Reading N Characters, BASIC, Model 100
ROOF-100	Rounding Off in BASIC, Model 100
RSWI-100	RS-232-C, What Is It?, Model 100
RUTP-100	RUNning a BASIC Program, Model 100
RWAT	Records, What are They?, Model 100
SACW-100	Strings, BASIC, Accessing Characters Within, Model 100
SBPT-100	Saving BASIC Programs on Cassette, Model 100
SCHO	SCHEDL, How to Get to, Model 100
SCIB-100	String Concatenation in BASIC, Model 100
SCMO	Switches and Connectors, Model 100
SCMU	Schedule Organizer, Using, Model 100
SFLO-100	Strings, Finding Length of, BASIC, Model 100
SGBA	Spaces, Generating in BASIC, Model 100
SHTG	Sound, How to Generate, Model 100
SHTU-100	Strings, BASIC, How to Use, Model 100
SIPS-100	Semicolon, In BASIC PRINTs, Model 100
SLTF	Saving and Loading TEXT Files, Model 100
SMLO	Saving a Machine Language Program, Model 100
SNER-100	SN Error, Model 100
SPHU-100	Single-Precision Variables in BASIC, How to Use, Model 100
SSFA-100	String Space, Finding Amount Left, Model 100
SSFO	Scratch, Starting From, Model 100
STAC-100	Stack, Description, Model 100
STER-100	ST Error, Model 100
STRD-100	STRING$ Command, BASIC, Model 100
SUBB-100	Subroutines, BASIC, Model 100
SUBP-100	Speeding Up Your BASIC Programs, Model 100
TCML-100	Transferring Control to a Machine Language Program from BASIC, Model 100

TCNU-100	Two's Complement Numbers, Using, Model 100
TEHT	TELCOM, How to Get to, Model 100
THTO	TEXT, How to Get to, Model 100
TIBP-100	Time, in BASIC Program, Model 100
TIMO	Time Interrupt, BASIC, Model 100
TMER-100	TM Error, Model 100
TMMO	Tokens in BASIC, Model I00
TOCH-100	Turning on the Computer, How To, Model 100
TSPD-100	To Stop the Display in BASIC, All Systems, Model 100
TSTP-100	To Stop the Program, Model 100
TUHO	TEXT, Using, Model 100
UEER-100	UE Error, Model 100
ULER-100	UL Error, Model 100
UTFB-100	Using Trigonometric Functions in BASIC, Model 100
VLFI	Variable Location in BASIC, Finding. Model 100

ACHT-100

Acoustic Coupler, How to Use, Model 100

Read procedure MWAT-100 if you know nothing about modems or acoustic couplers.

Also read procedure RHSO(100) to make certain that your communications parameters are set up properly for the data communications system you'll be calling.

The only time you'd want to use an acoustic coupler for telecommunications with the Model 100 is when you're at a remote location with a telephone without a standard telephone plug. Hotel telephones often have cables that go directly into the guts of the instrument, for example. If this is the situation, don't rip the phone out of the wall as they do on TV thrillers. You can still use the optional acoustic coupler for the Model 100 (RS 26-3805). One disadvantage of any acoustic coupler is that it's more prone to noise than a direct connect modem. Another disadvantage on the Model 100 is that you cannot automatically dial a number.

Follow these steps to use the acoustic coupler on the Model 100:

1. Set the DIR/ACP switch on the left-hand side of the Model 100 to ACP, or Acoustic Coupler.

2. Plug the connector on the acoustic coupler cable into the PHONE connector on the rear of the Model 100.

3. On the other end of the acoustic coupler cable are two cups. Slip the "microphone" cup over the microphone of the telephone receiver. The microphone end of the receiver is the end you talk into, the end with the cord. Slip the other cup over the receiver end of the telephone receiver, the end that you'd normally have next to your ear.

5. You're now ready to dial a Bulletin Board system, CompuServe, or other data communications system. Setup your modem this way:

   A. If you are originating the call, set the modem switch for ANS/ORIG on the left-hand side of the Model 100 to "Originate".

6. Enter TELCOM by procedure TEHT(100).

7. You do not have automatic dialing when using the acoustic coupler, therefore you must dial manually. Slip off the cup for the receiver. Dial up the number of the Bulletin Board or network while listening on the phone. After the phone is answered, you hear a pause followed by a high-pitched whine of the 'carrier' frequency. Take your time (you have a minute or so), and slip back the cup over the receiver portion of the handset.

8. Press Function Key 4 (TERM) to enter 'terminal mode.' You should see a new set of labels above the Function Key indicators:

```
Prev Down  Up  Full                    Bye
```

9. Press Function Key 4 (FULL/HALF) depending upon whether the communications system uses 'full' or 'half duplex'. Pressing Function Key 4 will 'toggle' the setting between full and half duplex. Most systems will use full duplex, so use that option if you aren't certain.

10. You should now see the prompt message of the Bulletin Board or network on the screen. If you do not, try typing ENTER a few times. If you still have trouble, review the steps above, and try again with another system (preferably). If you do see data on the screen, but it's garbled, go to procedure RHSO(100). If you see meaningful data, continue with the procedure for Bulletin Boards (BBUS-100) or CompuServe (CPSU-100).

ADFW-100

ASCII Characters, What Are They?, Model 100

ASCII refers to the code used to represent alphabetic, numeric, or special characters. It is a "7-bit" code, with the upper, or most significant bit, not used. This means that values of 0 through 127 are valid ASCII characters.

"ASCII", however, has been expanded to include 8-bit characters in many machines. The Model 100 can generate all 256 codes from 0 through 255 by either a single keypress or by combinations of two keys. Codes from 128 through 255 are graphics shapes and are always generated by the GRPH key plus another keyboard character. This means that all displayable characters can be generated from within BASIC or other programs. Codes for the Model 100 are shown in Table ADFW-100-1.

The codes lower than 32 decimal (**20H**) are called "control codes" because they are set aside for special functions such as "line feeds" (ejecting paper from a printer), "carriage returns" (start at beginning of line) or "move cursor". Although these codes can be generated by using the CTRL key and another key on the Model 100; by ESC; and by arrow keys, they will not display.

All the TRS-80 systems use ASCII in BASIC and other applications to represent alphabetic, numeric, and special characters.

Table ADFW-100-1 — *Model 100 ASCII Codes*

DECIMAL	HEX	DISPLAYED CHARACTER	KEYBOARD CHARACTER
Ø	ØØ		CTRL @
1	Ø1		CTRL A
2	Ø2		CTRL B
3	Ø3		CTRL C
4	Ø4		CTRL D
5	Ø5		CTRL E
6	Ø6		CTRL F
7	Ø7		CTRL G
8	Ø8		CTRL H
9	Ø9		CTRL I
1Ø	ØA		CTRL J
11	ØB		CTRL K

DECIMAL	HEX	DISPLAYED CHARACTER	KEYBOARD CHARACTER
12	ØC		CTRL L
13	ØD		CTRL M
14	ØE		CTRL N
15	ØF		CTRL O
16	1Ø		CTRL P
17	11		CTRL Q
18	12		CTRL R
19	13		CTRL S
2Ø	14		CTRL T
21	15		CTRL U
22	16		CTRL V
23	17		CTRL W
24	18		CTRL X
25	19		CTRL Y
26	1A		CTRL Z
27	1B		ESC
28	1C		←
29	1D		→
3Ø	1E		↑
31	1F		↓
32	2Ø		SPACEBAR
33	21	!	!
34	22	"	"
35	23	#	#
36	24	$	$
37	25	%	%
38	26	&	&
39	27	'	'
4Ø	28	(	(
41	29	)	)
42	2A	*	*
43	2B	+	+
44	2C	,	,
45	2D	-	-
46	2E	.	.
47	2F	/	/
48	3Ø	Ø	Ø
49	31	1	1
5Ø	32	2	2
51	33	3	3
52	34	4	4
53	35	5	5
54	36	6	6
55	37	7	7
56	38	8	8
57	39	9	9
58	3A	:	:
59	3B	;	;

DECIMAL	HEX	DISPLAYED CHARACTER	KEYBOARD CHARACTER
6Ø	3C	<	<
61	3D	=	=
62	3E	>	>
63	3F	?	?
64	4Ø	@	@
65	41	A	A
66	42	B	B
67	43	C	C
68	44	D	D
69	45	E	E
7Ø	46	F	F
71	47	G	G
72	48	H	H
73	49	I	I
74	4A	J	J
75	4B	K	K
76	4C	L	L
77	4D	M	M
78	4E	N	N
79	4F	O	O
8Ø	5Ø	P	P
81	51	Q	Q
82	52	R	R
83	53	S	S
84	54	T	T
85	55	U	U
86	56	V	V
87	57	W	W
88	58	X	X
89	59	Y	Y
9Ø	5A	Z	Z
91	5B	[	[
92	5C	\	GRPH -
93	5D	]	]
94	5E	^	^
95	5F	_	_
96	6Ø	`	GRPH [
97	61	a	A
98	62	b	B
99	63	c	C
1ØØ	64	d	D
1Ø1	65	e	E
1Ø2	66	f	F
1Ø3	67	g	G
1Ø4	68	h	H
1Ø5	69	i	I
1Ø6	6A	j	J
1Ø7	6B	k	K

DECIMAL	HEX	DISPLAYED CHARACTER	KEYBOARD CHARACTER
1Ø8	6C	l	L
1Ø9	6D	m	M
11Ø	6E	n	N
111	6F	o	O
112	7Ø	p	P
113	71	q	Q
114	72	r	R
115	73	s	S
116	74	t	T
117	75	u	U
118	76	v	V
119	77	w	W
12Ø	78	x	X
121	79	y	Y
122	7A	z	Z
123	7B	{	GRPH 9
124	7C	¦	GRPH —
125	7D	}	GRPH Ø
126	7E	~	GRPH [
127	7F		DEL
128	8Ø		GRPH p
129	81		GRPH m
13Ø	82		GRPH f
131	83		GRPH x
132	84		GRPH c
133	85		GRPH a
134	86		GRPH h
135	87		GRPH t
136	88	⋮	GRPH 1
137	89	√	GRPH r
138	8A	≠	GRPH /
139	8B	Σ	GRPH s
14Ø	8C	≈	GRPH ′
141	8D	±	GRPH =
142	8E	∫	GRPH i
143	8F		GRPH e
144	9Ø		GRPH y
145	91		GRPH u
146	92		GRPH :
147	93		GRPH q
148	94		GRPH w
149	95		GRPH b
15Ø	96		GRPH n
151	97	%	GRPH .
152	98	↑	GRPH o
153	99	↓	GRPH ,
154	9A	→	GRPH l
155	9B	←	GRPH k

DECIMAL	HEX	DISPLAYED CHARACTER	KEYBOARD CHARACTER
156	9C		GRPH 2
157	9D		GRPH 3
158	9E		GRPH 4
159	9F		GRPH 5
16Ø	AØ	′	CODE ′
161	A1	à	CODE x
162	A2	ç	CODE c
163	A3	£	GRPH 8
164	A4	‘	CODE "
165	A5	µ	CODE M
166	A6		CODE)
167	A7		CODE
168	A8		CODE +
169	A9		CODE s
17Ø	AA		CODE R
171	AB		CODE C
172	AC	¼	CODE D
173	AD	¾	CODE ;
174	AE	½	CODE /
175	AF	¶	CODE 0
176	BØ		GRPH 7
177	B1	Ä	CODE A
178	B2	Ö	CODE O
179	B3	Ü	CODE U
18Ø	B4	¢	GRPH 6
181	B5	~	CODE [
182	B6	ä	CODE a
183	B7	ö	CODE o
184	B8	ü	CODE u
185	B9	ß	CODE S
186	BA	™	CODE T
187	BB	é	CODE d
188	BC	ù	CODE ,
189	BD	è	CODE v
19Ø	BE	¨	CODE =
191	BF	ƒ	CODE F
192	CØ	â	CODE 1
193	C1	ê	CODE 3
194	C2	î	CODE 8
195	C3	ô	CODE 9
196	C4	û	CODE 7
197	C5	^	CODE -
198	C6	ë	CODE e
199	C7	ï	CODE i
2ØØ	C8	á	CODE q
2Ø1	C9	í	CODE k
2Ø2	CA	ó	CODE l
2Ø3	CB	ú	CODE j

DECIMAL	HEX	DISPLAYED CHARACTER	KEYBOARD CHARACTER
2Ø4	CC	ý	CODE y
2Ø5	CD	ñ	CODE n
2Ø6	CE	ã	CODE z
2Ø7	CF	õ	CODE .
2Ø8	DØ	Â	CODE !
2Ø9	D1	Ê	CODE #
21Ø	D2	Î	CODE *
211	D3	Ô	CODE (
212	D4	Û	CODE &
213	D5	Ï	CODE I
214	D6	Ë	CODE E
215	D7	É	CODE D
216	D8	Á	CODE Q
217	D9	Í	CODE K
218	DA	Ó	CODE L
219	DB	Ú	CODE J
22Ø	DC	Ý	CODE Y
221	DD	Ù	CODE <
222	DE	È	CODE V
223	DF	À	CODE X
224	EØ		GRPH Z
225	E1	▘	GRPH !
226	E2	▝	GRPH @
227	E3	▖	GRPH #
228	E4	▗	GRPH $
229	E5	▚	GRPH %
23Ø	E6	▞	GRPH "
231	E7	▔	GRPH Q
232	E8	▁	GRPH W
233	E9	▌	GRPH E
234	EA	▐	GRPH R
235	EB	▛	GRPH A
236	EC	▜	GRPH S
237	ED	▙	GRPH D
238	EE	▟	GRPH F
239	EF	█	GRPH X
24Ø	FØ	┌	GRPH U
241	F1	─	GRPH P
242	F2	┐	GRPH O
243	F3	┬	GRPH I
244	F4	├	GRPH J
245	F5	│	GRPH :
246	F6	└	GRPH M
247	F7	┘	GRPH >
248	F8	┴	GRPH <
249	F9	┤	GRPH L
25Ø	FA	┼	GRPH K
251	FB	◤	GRPH H
252	FC	◢	GRPH T
253	FD	◥	GRPH G
254	FE	◣	GRPH Y
255	FF	▒	GRPH C

ADMO

Address Organizer, Using, Model 100

The Address Organizer ADDRSS is really nothing more than an applications program that is geared to searching for strings of characters and displaying them on the screen. It operates similarly to Search in TEXT (see TUHO, this section), but displays as many occurrences of the string as it finds, on a line basis.

Suppose that you had the file ADRS.DO, generated as a list of names, addresses, and telephone numbers. It looks like this:

```
Whytset-Jones, Percy, CBE, 1234 Hedley,
Twinkley-on-Thames, Whitcombshire, Great Britain
:1-Ø32-343-876-8877:

Ralston, Billy Joe Bob, 111 East Texan Forever Drive,
Fort Worth, TX 761Ø2 :1-817-555-1212:

Cheddar, M., 444 North Compasset, Radical, WI 53181
:1-414-555-1212:
```

. . . plus many more.

The file consists of a number of entries. Each entry is terminated by an ENTER and consists of one or more lines. Another word for the entry is a "record". See RWAT(100).

You could easily use ADDRSS to find and list the telephone number of Billy Joe Bob Ralston by "Finding" Billy Joe Bob (a somewhat unique name), Finding all Fort Worth addresses, or Finding all 76102 Zip Code lines.

Like SCHEDL, there's no mystique about ADDRSS. It's simply a way to go through a large file of addresses called ADRS.DO and pull out the ones that you're interested in, grouped by special symbol or date. It's akin to searching through a stack of 3 inch by 5 inch index cards on which you have names, addresses, and telephone numbers. The ADDRSS program is the device by which all of the appropriate cards are pulled out and laid down in a row for you to see.

See AHTO(100) to find out how to get into ADDRSS.

Use a colon to bracket telephone numbers: As any ADRS.DO file can be used to automatically dial up the number by the TELCOM program in the Model 100, and because TELCOM assumes a telephone number is bracketed by colons, always use this format if there's any chance you'll be using the auto dial feature (see DNAO, this section).

Use a dash between area codes, etc: Again, the format is established by TELCOM. If you'll never be using TELCOM to dial numbers from the ADRS.DO file, don't worry about this, but otherwise use a dash where you would normally put a space.

To find information in an existing ADRS.DO file:

1. After the

```
Schd:
```

prompt, press F1.

2. You'll now see

```
Schd: Find
```

3. Enter a string of characters to be found. If you wanted a list of all 76102 Zip Code telephone numbers, for example, you might enter 76102. In this example, you'd then see:

```
Schd: Find 761Ø2
Ralston, Billy Joe Bob, 111 East Texan Forever Drive,
Fort Worth, TX 761Ø2 :1-817-555-1212:
```

Of course, if there were other addresses with a 76102 Zip Code, you'd see a number of them listed at the same time.

The case — upper or lower — is ignored in the search.

4. At this point either all lines with the "Find" string have been displayed, or there are too many lines to display on the screen. In the latter case, you'll see the message:

```
More Quit
```

above Function Key indicators F3 and F4. To see more of the list, press Function Key F3. To quit, press Function Key F4.

5. When all items have been displayed, the prompt message:

```
Schd:
```

will reappear.

As a second example of searching, suppose that you had wanted to find someone's name, but couldn't remember it, although you knew that he lived on a street that started with "Com". Entering

```
Schd: Find Com
```

would have resulted in

```
Whytset-Jones, Percy, CMBE 1234 Hedley,
Twinkley-on-Thames, Whitcombshire, Great Britain
:1-Ø32-343-876-8877:
Cheddar, M., 444 North Compasset, Radical, WI 53181
:1-414-555-1212:
```

All entries with the character string "COM", "com", "Com" or any combination of upper and lower case characters were found.

To scan the entire ADRS.DO file:

1. After the

```
Schd:
```

prompt, press F1.

2. You'll now see

```
Schd: Find
```

3. Press ENTER. This essentially says, find *everything*.

4. The first lines of the ADRS.DO file will then be displayed. At this point either all lines of the ADRS.DO file have been displayed, in a short file, or there are too many lines to display on the screen. In the latter case, you'll see the message:

```
More Quit
```

above Function Key indicators F3 and F4. To see more of the list, press Function Key F3. To quit, press Function Key F4.

5. When all items have been displayed, the prompt message

```
Schd:
```

is again displayed.

To list items on the printer: Do the same thing as in the two procedures above, but press Function Key F5 (LFND) instead of Function Key F1 (FIND). The list will be printed on the system line printer without pauses.

To update the ADRS.DO file: Use the TEXT program operating on File ADRS.DO. See TUHO(100).

AEBP
Auto Execute of a BASIC Program, Model 100

To automatically execute a BASIC program whenever power is turned on, do this:

1. Load BASIC (see BHTO, this section).

2. Construct or load the BASIC program for automatic execution.

3. SAVE the program from BASIC.

4. Execute:

```
IPL"name"
```

where "name" is the program name.

5. Turn off the power.

6. When power is again turned on, the program named under IPL will execute. The IPL stands for "Initial Program Load".

7. As long as you remain in BASIC with the IPL program loaded, powering off and then on will automatically execute the IPL program.

AFWA-100
ASCII Files, What Are They?, Model 100

Read ADFW-100 if you don't know what ASCII characters are.

ASCII files are made up of ASCII characters from 32 decimal (**20H**) through 127 decimal (**7FH**), in addition to special control codes such as carriage return (**0DH**). No other codes are used, in most cases. This means that the file is "displayable" on the video screen or "printable" on the system line printer. ASCII files take up more space than an "encoded" type of file, but can easily be examined by display or printing.

Model 100 BASIC and other programs work with a specially encoded type of file that saves space. One BASIC "token" byte (see TMMO, this section), for example, may replace six bytes or more of a command name. These programs, however, sometimes also offer the option of writing ASCII files. The ASCII files use only ASCII characters, and no special codes.

ASCII files are a "standard" format, and for this reason, certain Model 100 commands, such as MERGE, will only work with ASCII files.

AGTU-100
Arrays in BASIC, Using, Model 100

Read procedure AGTU in the main section. The information there is identical to the Model 100 use.

AHTO
ADDRSS, How to Get to, Model 100

Before you can access ADDRSS successfully, you must have created a file called ADRS.DO by using TEXT. The ADRS.DO file is a collection of names and address entries that you have jotted down — any data that you care to record. The ADRS.DO file will appear as a file name on the main menu.

If you have the main menu on your screen, you'll see the words "BASIC TEXT TELCOM ADDRSS" on the second line of the screen. Position the shadow area, the cursor, over the word ADDRSS by using the arrow keys to the upper right of the keyboard. Now press ENTER. You should now be in ADDRSS and see the display:

```
Adrs:
```

Read procedure ADMO(100) to find out how to use ADDRSS.

ALIB-100
Inserting a Line in BASIC, Model 100

BASIC will insert a new line in numerical order in a BASIC program in RAM if you simply type a new line with a line number that corresponds to the insertion point. For example, to insert a line between BASIC program line 100 and 112, simply type a new line with line number 111.

You may also use the TEXT editor (see TUHO in this section), but this is overkill for simple BASIC line insertion.

ALWI-100
Assembly Language, What is It?, Model 100

Read MLWI-100, "Machine Language, What is It?, Model 100".

Now read procedure ALWI in the main section, bearing in mind that assembly language for the Model 100 will be similar, but not identical to the example in the procedure.

At this time of writing, there is no assembler for the Model 100. There certainly will be, however, as the "hooks" (BASIC CALL command and others) are in Model 100 BASIC.

Assembly language programming for the Model 100 is completely feasible. There are many assemblers for the 8085 microprocessor, and they could very easily be adapted for the Model 100.

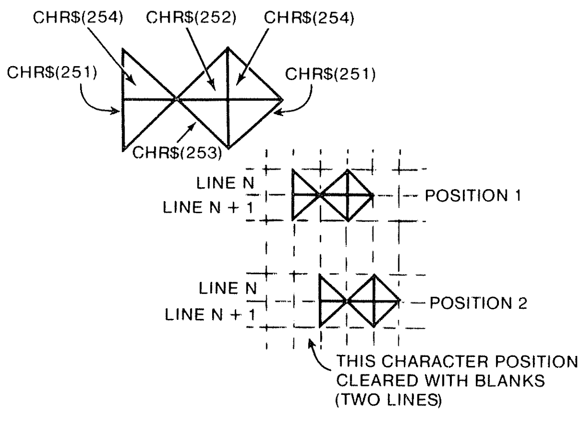
ANON
Animation, Model 100

Read GHSO for explanations of high-speed graphics. Normal SET/RESET graphics are not fast enough for effective animation.

The fastest graphics are done by assembly language, and it may benefit you to learn assembly language if you have more than a casual interest in graphics.

If you are working in BASIC, use the string method explained in GHSO to move predefined figures. One of the best methods is described in GHSO, where a single string defines an entire figure. The figure can easily be moved around by using a moving starting position for the string and drawing the string at a series of locations, erasing any old pattern, as shown in Figure ANON-1.

Figure ANON-1 — *Animation in the Model 100*



```
100 ' FISH FIGURE
110 CLS
120 A$=CHR$(254)+CHR$(252)+CHR$(254)
130 B$=CHR$(251)+CHR$(253)+CHR$(251)
140 FOR X=200 TO 235
150 PRINT @X-1,"   ";:PRINT @X+39,"   ";
160 PRINT @280,CHR$(13)
170 PRINT @X,A$;
180 PRINT @X+40,B$;
190 FOR Z=0 TO 30:NEXT Z
200 IF RND(1)>.5 THEN 220
210 PRINT @X-37,"o";
220 NEXT X
230 GOTO 110
```

AOIB-100

Arithmetic Operations in BASIC, Model 100

Read procedure AOIB in the main section. Arithmetic operations in the Model 100 are identical to the procedures discussed in AOIB.

notes

BBUS-100
Bulletin Boards, Using, Model 100

Read the first part of procedure BBUS in the main section for general background on bulletin boards.

Read procedure MHTU-100 or ACHT-100 for the procedure in dialing up bulletin boards. Use the ADRS.DO file to auto-dial the bulletin board number.

BHTO
BASIC, How to Get to, Model 100

If you have the main menu on your screen, you'll see the words "BASIC TEXT TELCOM ADDRSS" on the second line of the screen. Position the shadow area, the cursor, over the word BASIC by using the arrow keys to the upper right of the keyboard. Now press ENTER. You should now be in BASIC and see the display:

```
TRS-8Ø Model 1ØØ Software
Copr. 1983 Microsoft
XXXXX Bytes free
```

Congratulations, you're in BASIC, and the world's your oyster.

BPIB-100
Breakpointing in BASIC, Model 100

Read procedure BPIB in the main section; the information there applies to breakpointing BASIC in the Model 100 as well.

BSEC-100
To Backspace and Erase Character, Model 100

Use the DEL BKSP key (without SHIFT) or the left arrow key.

BSEL-100
To Backspace and Erase Line, Model 100

To completely erase the current line, press SHIFT followed by BREAK/PAUSE.

BSER-100
BS Error, Model 100

See BSER in main section.

notes

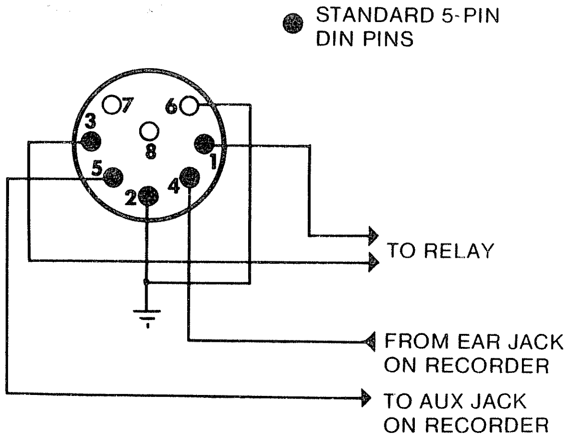
CCAM-100
Cassette Connector, Model 100

The cassette connector on the Model 100 is a "superset" of the connector for the Models I, III, and Color Computer. If you are wiring your own connector, use the "thin-wall" (metal) version of a standard 5-pin DIN male audio plug (Radio Shack 274-003) to fit the cassette connector.

Refer to Figure CCAM-100-1. Pins 1 and 3 connect to an internal relay and can be used as a programmable switch for low-voltage, low-current applications. Do not use for over 12 volts dc and 1/2 ampere or so. If you connect a milling machine to these pins, **you** will be responsible for the results. These pins normally control the cassette REMote input to turn the recorder on and off.

Pins 2 and 6 are signal ground. Pin 4 is the input to the computer from the EARphone jack of the recorder. Pin 5 is the output from the computer to the AUXiliary input of the recorder. Pins 7 and 8 are not connected. See CRPI-100 for a description of cassette recorder plugs.

Figure CCAM-100-1 — *Cassette Connector Pins*



CCAM
CFCS
CFOW
CHTP
CLTD

CFCS-100
Converting the First Character of a String to Numeric in BASIC, Model 100

Read procedure CFCS in the main section. Model 100 use of ASC is identical.

CFOW-100
Cursor, Finding Out Where It Is In BASIC, Model 100

Suppose you're doing a general input operation in BASIC and want to find out the current cursor position. The POS function will tell you where that blinkin' light is on the row:

```
1ØØ A=POS(Ø)
```

stores the current cursor position along the row in variable A. The "(0)" is a "dummy" argument that really doesn't do anything.

The position will correspond to the character position along the row and will be 0 through 39.

POS is handy for columnating data (see CHTP-100) or for word-processing applications.

The CSRLIN command will tell you the line number of the cursor:

```
2ØØ B=CSRLIN
```

This function doesn't need a dummy argument and returns a line number of 0 through 7.

CHTP-100
Columns, How to Put Things In, in BASIC, Model 100

Read procedure CHTP in the main section. Columnating data for the Model 100 in BASIC is identical to the way it's done in the other machines. Print zones for commas are at character positions 0 and 14, with the first position numbered 0. TABs can be done from 0 through 255 with TAB(0) through TAB(39) going onto one line. PRINT USING can be used as in other systems.

CLTD-100
Clearing the Display in BASIC, Model 100

To clear the display while in BASIC, enter CLS.

This action clears the display and positions the cursor to the upper-left corner after an "OK". Any BASIC program remains intact.

CMLO
Calling a Machine Language Program in BASIC, Model 100

If you want to interface to a machine language program from BASIC, follow these steps:

1. Assemble or translate the machine language code to be loaded so that it will run properly at the location in RAM memory you desire. This is the hardest part of the task. Currently there is no assembler for the Model 100. Look to books on 8080/8085 assembly language to tell you how to do this.

2. Protect the memory into which the machine language program is to be loaded by the procedure given in PROT–100.

3. Put the machine language code into the area of RAM memory required. This can be done by loading a machine language file (see LMCO or LMRO in this section), by POKEing the machine language code (see PPKU–100), or by other means.

4. Write your BASIC program so that it includes the following statements:

```
11Ø CALL MMMMM,A,HLHLH
```

In the statements above, MMMMM is the address of the start of the machine language code in decimal. This is the area in RAM memory into which the machine language code is located. The optional "A" operand is a numeric constant, variable, or expression from 0 through 255. It denotes the value that will be put into the 8085 A register before the CALL is performed. The optional "HLHLH" operand is a numeric constant, variable, or expression from –32768 through 65535. It denotes the values that will go into the HL register before the CALL is made. The two optional operands are only used to transfer "parameters" to the machine language subroutine if required (see PVMO in this section). The CALL will take the machine language address MMMMM and "call" that location, saving the return point of the next BASIC statement after the USR call.

Here's an example: We have a simple assembly language program to fill the first 256 character positions of the screen with asterisks, shown in Figure CMLO-1. This has been translated to machine code decimal values as given in procedure CFHD(main sequence), and incorporated into BASIC DATA statements as shown below. The BASIC code to relocate the machine code and to call the machine language program is shown below. (Note: The screen area will be updated after BASIC program execution.)

```
9Ø CLEAR 1ØØ,59999
1ØØ REM DEFINE ML CODE
11Ø DATA 6,Ø,33,Ø,254,54,42,5,2ØØ
12Ø DATA 35,195,1Ø1,234
13Ø REM MOVE THE CODE TO 6ØØØØ
14Ø FOR I=6ØØØØ TO 6ØØØØ+12
15Ø READ A:POKE I,A
16Ø NEXT I
17Ø REM CALL THE ML SUBROUTINE AT 6ØØØØ
18Ø CALL 6ØØØØ
19Ø REM RETURN AT THIS POINT
```

Figure CMLO-1 — *Screen Program in Assembly Language*

```
; 8Ø85 AL PROGRAM TO FILL 256 ASTERISKS TO SCREEN BUF
6ØØØØ Ø6 ØØ              MVI   B,Ø        ;Ø TO B REGISTER
6ØØØ2 21 ØØ FE           LXI   H,SCREEN   ;SCREEN BUF TO HL
6ØØØ5 36 2A      LOOP    MVI   M,'*'      ;ASTERISK TO SCREEN
6ØØØ7 Ø5                 DCR   B          ;DECREMENT COUNT
6ØØØ8 C8                 RZ               ;RETURN IF DONE
6ØØØ9 23                 INX   H          ;INCREMENT HL
6ØØ1Ø C3 65 EA           JMP   LOOP       ;CONTINUE
```

CMWI
Command Mode, What Is It, Model 100

Most BASIC commands can be executed after the OK prompt in BASIC. If, for example, you want to output characters to the line printer, you could do

```
OK
LPRINT CHR$(27);CHR$(31);
```

or you could do

```
OK
FOR I=31ØØØ to 31ØØ5:PRINT PEEK(I),:NEXT I
```

Executing BASIC commands **immediately** in this fashion often is a great convenience, as you don't have to alter any BASIC program.

CNER-100
CN Error, BASIC, Model 100

See CNER in main section.

CPCA-100
Centronics (Printer) Cables, Model 100

The 26 pin-connector on the rear of the Model 100 uses the same Centronics bus signals as other Radio Shack systems, but they are arranged in a different configuration. Read procedure CPCA in the main section, but bear in mind that the Model 100 printer "pin-out" is different. I'd advise buying the standard Radio Shack cable for printer connections.

CPSU-100
CompuServe, Using, Model 100

Read the first part of procedure CPSU in the main section for general background on CompuServe.

Read procedure MHTU-100 or ACHT-100 for the procedure in dialing up CompuServe. Use the ADRS.DO file to auto-dial your local CompuServe number.

Read the latter part of procedure CPSU for information on CompuServe actions once you have reached the service.

CRPI-100
Cassette Recorder Plugs, Insertion of, Model 100

Figure CRPI-1 (main section) shows the correct insertion of cassette recorder plugs on the typical cassette recorders for the Model 100.

A good tip: Remember the mnemonic device "black is back"; the large black plug goes into the jack furthest to the rear of most recorders.

CSHT-100
Comparing Strings in BASIC, Model 100

Read procedure CSHT. It applies equally well to the Model 100.

CSNV-100
Converting BASIC Strings to Numeric and Vice Versa, Model 100

Read procedure CSNV in the main section. The use of VAL and STR$ is the same in the Model 100.

CTLC-100
Cassette Tape Loading Difficulties, Model 100

If you have difficulties loading a cassette tape file, try these cures:

1. Make certain the tape is positioned properly by removing the EAR output plug and REM plug and listening to the file. Position the tape right before the file and try again.

2. Temporarily remove the AUX input from the cassette recorder and try again.

3. Check the volume control setting. Try various levels.

4. If you are near a monitor or television receiver, physically relocate the cassette recorder away from the television or monitor as far as possible. Try either side of the television or monitor; there may be interference from the flyback transformer or other electronics.

5. Are you using a high-quality tape? It may pay to get a "certified" cassette tape such as Radio Shack's 26-301 or 26-302. Certified tapes have been tested to ensure that there are no "dropouts" — points on the tape where the magnetic material is thin or non-existent.

Recommended volume setting for Model 100: 7 on scale of 10.

CUSE-100
CHR$ Use, BASIC, Model 100

See procedure CUSE in the main section for a discussion of CHR$; operation is identical on the Model 100.

notes

DDER-100
DD Error, Model 100

See DDER in main section.

DEWI
"DO" Extension, on Model 100 Files, What Is It?

The file name extension ".DO" stands for "Document" and means that the file is a text file (see AFWA-100).

DFMO
Date, Finding Out Today's, Model 100

Method One: The menu display has the date, day, and time on the first line. See DIBO(100) and TIBP-100 to set these parameters.

Method Two: Use DATE$ in BASIC, and the current date will be returned. You can use it as any other character string:

```
100 A$=DATE$          get date
```

Once entered, the date, day, and time will remain accurate unless MEMORY POWER is turned off (SSFO in this section) or unless a cold start is done.

DIBO
Date, in BASIC Program, Model 100

The DATE$ function gets the date in the format MM/DD/YY. The current date, of course, must be valid; this means that the date must have been entered sometime previously by DATE$="MM/DD/YY" either in the command mode or in the program. The date is automatically changed at midnight. (If you're in Milwaukee, you might consider staying up for the event...) Note that even when the power switch is off, the Model 100 updates both time and date.

To reset the date on the Model 100, go to BASIC and then enter the current date in this format:

```
DATE$="MM/DD/YY"
```

where MM, DD, and YY are the month, day, and year, each in two digits (use a leading 0 if necessary, as in 01/02/82).

The DAY$ function gets the current day of the week. It's a 3-character string of Mon, Tue, Wed, Thu, Fri, Sat, or Sun. Again, enter the current day by DAY$="DDD". Again, the day of the week is updated at (ominous sound of clock. . .) MIDNIGHT!

DLBO
Drawing Lines and Boxes, Model 100

See GHSO(100) for information on high-speed graphics.

LINE: LINE should be called, "LINEBOXFILLED-INBOX".

LINE will draw a line between any two points, as in

```
100 LINE (20,60)-(40,50)
```

which draws a straight line between graphics points 20,60 and 40,40 (see MOPA in this section).

The average line is drawn in about 54 milliseconds and the worst case is about 100 milliseconds, about 20 times faster than the fastest BASIC line drawing routine.

There is an optional parameter which can be used at the end of the LINE command. If the parameter is an odd value, the points of the line will be set; if the parameter is an even value, the points of the line will be reset:

```
100 LINE (20,60)-(40,50),1      'set line
100 LINE (20,60)-(40,50),0      'reset line
```

LINE will also draw a box (rectangle) outline. In this case the two coordinates specify the opposing corners of the box.

```
100 (50,50)-(60,60),1,B     'B specifies box
100 (50,50)-(60,60),0,B     'reset box
```

The box is drawn at speeds comparable to drawing four lines.

A third use is in drawing a "filled-in" box. The filled-in box is, of course, a lot slower (the time may go over ten seconds for large boxes), but still excellent for such a powerful command:

```
100 (50,50)-(60,60),1,BF    'BF is filled-in box
```

Using an odd "switch" value resets the points in a filled-in box.

DLIB-100
Deleting Lines in BASIC, Model 100

For a single line: Type in line number alone. Line will be deleted.

For multiple lines: Use the Edit mode in BASIC. To get into the Edit Mode while in BASIC, enter

```
EDIT
```

This will bring you into the TEXT Editor, and you can edit using the commands in TUHO(100). To get back to BASIC, press Function Key F8. If you want to edit only a portion of the BASIC program enter

```
EDIT line1-line2
EDIT -line2
EDIT line1-
```

These commands establish a "range" of lines for the edit. The first edit is of line1 through line2, the second is of every line from beginning to line2, and the last is from line1 through the end of the program.

Here's one kicker in using the TEXT Editor: The TEXT editor works with ASCII (see ADFW-100) while the BASIC interpreter works with BASIC "tokens" (see TMMO in this section). When going from the BASIC interpreter to the Text Editor, the BASIC lines are converted to ASCII; when re-entering the BASIC interpreter from the text editor, the ASCII is reconverted. This is all done smoothly, but it does take some time, which might be apparent for longer BASIC programs by the "WAIT" message on the last line of the display.

DNAO

Dialing a Number Automatically, Model 100

You can dial any number from an ADRS.DO file. Follow the steps in procedure MHTU under **Dialing a number from a RAM file**. In this case you'll be searching for a number without a following "<>", but with colons at the front and back of the number. (The "<>" causes entry into the "Terminal Mode", and you don't want that.) When you've found the number in the ADRS.DO file, simply press Function Key 2 (CALL); you'll see the number displayed as it is dialed. Before the number completes, pick up the phone and you'll be connected.

This procedure is handy for dialing many numbers that are contained in the ADRS.DO file. Almost as handy as dialing them from an automatic dialing machine. . .

DNMO

Dialing a Number Manually, Model 100

To dial a telephone number using TELCOM:

1. Hook up the Modem Cable as described in MHTU-100.
2. Get to TELCOM by TEHT(100).
3. Press Function Key 2 (CALL).
4. After the Call prompt, type the number to dial, followed by ENTER.
5. You'll see the numbers displayed after the message "Calling".
6. If you don't pick up the phone before the end of the call, you'll be disconnected.
7. Why not try just **dialing** the number from your Touch-tone phone?

DPHU-100

Double-Precision Variables in BASIC, How to Use, Model 100

Double-precision variables are the "default" variable type. If you do not explicitly define a variable as integer (see IVHU-100), single-precision (see SPHU-100), or string (see SHTU-100), then it is a double-precision variable.

Double-precision variables use a similar floating-point implementation as single-precision variables (SPHU-100), but extend the number of decimal digits to 14. Double-precision variables take up 8 bytes of storage and should be used only when great accuracy is required as storage and processing time is greater than other types of variables.

Specify a double-precision variable after a DEFSNG, DEFINT, or DEFSTR by the suffix "#". A# and BB# are double-precision variables:

```
90 DEFSNG A-C
100 A#=1234.4444444445
110 BB#=234/.09888
```

A range of double-precision variables may be specified by the DEFDBL command. DEFDBL A-G, for example, specifies all variables starting with A through G as double precision; AS, FD, and GG would be double precision in this case.

Using a suffix of "D" also denotes a double-precision number, in this case with scientific notation (see AOIB).

```
100 A=123456.789D+17
```

Bear in mind again that you don't have to use the "#" suffix to specify double precision. If you don't use any suffix, you'll automatically get a double-precision variable.

DSPO

Dumping the Screen to the Printer in BASIC, Model 100

Executing

```
LCOPY
```

will "dump" the current text on the screen to the system line printer. Characters that are not printable characters will be ignored.

Pressing the PRINT key will do the same thing.

DSRC-100
DATA Statements and Related Commands, BASIC, Model 100

Read procedure DSRC in the main section; the information on DATA statements and related commands applies exactly to the Model 100.

DZER-100
Divide by Zero Error, Model 100

See DZER in main section.

notes

EFAF
8085 Address Format, Model 100

Confused about storing CALL addresses or getting string addresses? It isn't bad enough that you have to convert between decimal and hexadecimal, but then the darn things are reversed! Sorry, that's the way the 8085 microprocessor in the Model 100 works. It looks for an address value least significant byte followed by most significant byte. If you had these values in two memory locations,

```
1ØØ,128
```

then the memory address represented would be 128 (most significant byte), 100 (least signicant byte) or 128*256+100 or 33,024 decimal (see CFHD in the main section). *All two-byte memory addresses are arranged in that fashion,* and you'll have to store them this way for machine language functions.

EMBO
Edit Mode, BASIC, Model 100

See procedure TUHO(100) for use of the Model 100 Text Editor.

To enter TEXT from BASIC use

```
EDIT
EDIT line1-line2
EDIT -line2
```

or

```
EDIT line1
```

EDIT without arguments will edit the entire program. EDIT with line numbers will edit a "range of lines". The minus sign stands for "all lines from the beginning" (-line2) or "all lines to end" (line1-). To return to BASIC at any time in the TEXT edit, simply press Function Key F8.

You'll see a "WAIT" message when going back to BASIC from TEXT as the ASCII file in TEXT is reconverted to BASIC tokens (see TMMO in this section). This is normal.

EPHT-100
Ending a BASIC Program, How to, Model 100

Read procedure EPHT in the main section; the information there applies to the Model 100 as well.

ETIB-100
Error Trapping in BASIC, Model 100

Read ETIB in the main section. All commands except ERR$ (Model II) apply.

EYBP-100
Erasing Your BASIC Program, Model 100

Enter NEW. NEW "reinitializes" the BASIC interpreter "pointers", in effect erasing your program. The string space allocation is kept the same as established in CLEAR (see OSER-100).

notes

FCER-100
FC Error, Model 100

See FCER in main section.

FKDB
Function Keys, Defining, BASIC, Model 100

Want to define the 8 Function Keys to automatically generate a string of up to 15 characters when pressed in BASIC? Do a

```
KEY n,"string"
```

Thereafter, every time the nth Function Key is pressed the string of characters "string" will be generated, just as if you had typed them in from the keyboard.

```
KEY 3,"?DAY$,DATE$"+CHR$(13)
```

for example, will display

```
?DAY$,DATE$, followed by ENTER
```

after you press Function Key 3 in BASIC. (The question mark is an abbreviation for PRINT.) This will let you print out the day of the week and date with a simple keypress.

You can redefine the keys at will by new KEY commands.

To reset all keys, do

```
CALL 23164,0,23366
CALL 27795.
```

What is this CALL action? The CALL calls a ROM subroutine (see RCWA-100) in the BASIC interpreter that will reset the key definitions.

To see the current key definitions, do

```
KEY LIST
```

FKMO
Function Key Interrupts, BASIC, Model 100

The Model 100 contains 8 Function Keys that can be activated for "interrupts". From 1 to 8 of the Function Keys can be linked to a BASIC subroutine so that any time one of the Function Keys is pressed, the corresponding BASIC subroutine is entered. This enables the user to define his own "labels" and functions for the Function Keys, depending upon his BASIC applications program. You could define Function Key 1 as "add entry to a mailing list", Function Key 2 as "delete mailing list entry", and so forth.

If you don't know how a "computed GOSUB" works, read procedure OGHU-100. The command for defining one or more Function Keys is

```
ON KEY GOSUB line#1, line#2,...,line#8
```

It works similarly to an ON n GOSUB. One to 8 line numbers correspond to Function Keys 1 through 8. Use commas if the keys are undefined.

```
100 ON KEY GOSUB 1000,2000,1050,,3000
```

defines the subroutine starting at line number 1000 for Function Key 1, the subroutine starting at line number 2000 for Function Key 2, the subroutine starting at line number 1050 for Function Key 3, nothing for Function Key 4, the subroutine starting at line number 3000 for Function Key 5, and nothing for Function Keys 6 through 8.

Once the Function Keys have been "defined", you must enable them by a

```
KEY ON
```

BASIC command. Thereafter, any time a Function Key is pressed during any portion of a BASIC program, there'll be an automatic transfer to the corresponding Function Key subroutine.

To disable all ON KEY interrupts, do

```
KEY OFF
```

To enable all again, do

```
KEY ON
```

To enable or disable individual keys, do

```
KEY (n) ON
KEY (n) OFF
```

where n is the key number, 1 through 8.

To disable the ON KEY interrupts, but cause the Model 100 to remember that the interrupt occurred, do

```
KEY STOP
```

When the interrupt is again enabled by KEY ON, the KEY STOP command causes an immediate GOSUB action to the ON KEY subroutine. Suppose that you had the Function Keys defined, but then disabled them by KEY OFF. If you pressed one or more of the keys, and then enabled them by KEY ON, nothing would happen. However, if you had disabled the Function Keys by KEY STOP, pressed a key, and again enabled the keys by KEY ON, an immediate GOSUB for the pressed key would occur after the KEY ON.

Note that changing the Function Key definitions via ON KEY GOSUB alters the "normal" definitions, such as getting back to the menu by pressing Function Key 8, and so forth.

FMOU
Files, BASIC, Model 100, Using

Data files may be constructed, read, and updated in BASIC on the Model 100. These BASIC files are a type of file known as "sequential" files. Sequential files are easy to use, easy to decode on LISTings and other inspection methods, and eas(ier) to understand than other types of files. On the other hand, sequential files are just that — sequential. When you access sequential files you generally start at the beginning of the file and then go through from beginning to end to find the record you're looking for. This is fine for certain types of processing, merging alphabetized records in alphabetized files, for example, but time consuming for other operations, finding random records from a master file of part numbers, for instance.

Parts of a sequential file: Sequential files contain "records". Each record may be of "variable length", although this is not absolutely necessary. There may be any number of records in a file, within the constraints of the data area available. Generally, a record is related to some organized unit of data. A record in an inventory file, for example, might have a part number, manufacturer's part number, description, number on hand, number ordered, and so forth.

General flow for creating a sequential file: To create a sequential file, do this:

1. Execute a BASIC OPEN statement. The format of OPEN is

```
1ØØ OPEN "device" FOR OUTPUT AS file#
```

Model 100 files may go to RAM, CASsette, COMmunications line, LCD (display), LPT (line printer), or MDM (modem), and the "device" parameter specifies which of these devices is to be used. "OUTPUT" specifies that data is to be output to a new file. The file# is a number of 1 through n, a file "number" or "buffer" associated with the file.

```
1ØØ OPEN "RAM:INVENT.DO" FOR OUTPUT AS 1
```

for example, "opens" a file called INVENT.DO in RAM for sequential OUTPUT from file number or buffer 1.

There's nothing mysterious about OPENing a file. The operation just makes an entry in the RAM directory of files, putting in the name, spec'ing it as a sequential file, and finding a vacant space in the RAM for the file to use. At this point the system does not know how large the file will be, and doesn't make any assumptions about the disk area required.

OPENing a file for cassette works similarly to OPENing one for RAM.

```
OPEN "CAS:INVENT" FOR OUTPUT AS 1
```

for example, OPENs a file called INVENT for output, using buffer 1.

OPENing a file for the communications line or modem is similar, except that the configuration may have to be specified as well:

```
OPEN "COM:66Ø1E" FOR OUTPUT AS 1
```

The file number (buffer number) is simply the buffer to be used to collect the data for the file. The buffer number defines which one of several buffer areas are to be used with the file. Look upon the buffer as a central collection point where the data is accumulated until a large enough chunk is collected to be sent to RAM or another device.

2. Now that the file is OPENed, you can write to it. You may do as many writes as you want. Each write is very similar to a PRINT statement; as a matter of fact, a PRINT#1 will print to the sequential file in lieu of PRINTing on the display. The operation is very similar but the ASCII (character) data goes out to the RAM, CAS, COM, LCD, LPT, or MDM file in lieu of PRINTing. The number for the PRINT corresponds to the buffer number. PRINT#3 writes out to buffer 3.

Just as you could specify a list of PRINT items for display or line printer, you can specify a list of PRINT# items. Using the inventory example, you could store a part number, manufacturer's part number, description, number on hand, and number ordered, as follows:

```
1ØØ OPEN"RAM:INVENT.DO"FOR OUTPUT AS 1
11Ø INPUT PN,MP,DE$,NH,NO
12Ø IF PN=-1 THEN GOTO 15Ø
13Ø PRINT#1,PN,MP,DE$,",",NH,NO
14Ø GOTO 11Ø
15Ø ...
```

We've done some really clever things here, like assigning names that match the item types, and so forth, in addition to looking for a part number of –1 to denote the end of the entry.

The only thorny part in the write is the way string data is handled. As the file will use spaces between data items and the string data may contain spaces, a comma is used to separate the string from the other data items. The computerese word for this is "delimiter" — the comma "delimits" the end of the string.

3. When the terminating part number (–1) is entered, the file write is over. However, at this point there may still be data in the buffer, as the buffer may not yet have been filled up from the last "dump" to the device. The CLOSE statement closes this buffer out by writing any remaining data to the file device (in some circles this is called "flushing the buffer"); it also notifies the system software that the file has been completely written, and this results in a possible directory entry (as for RAM) being completed, with the number of records in the file and other information.

The complete inventory program now looks like:

```
1ØØ OPEN"RAM:INVENT.DO" FOR OUTPUT AS 1
11Ø INPUT PN,MP,DE$,NH,NO
12Ø IF PN=-1 THEN GOTO 15Ø
13Ø PRINT#1,PN,MP,DE$,",",NH,NO
14Ø GOTO 11Ø
15Ø CLOSE 1
```

The CLOSE statement "closed" the file associated with buffer number 1, our INVENT.DO file.

4. This general sequence of OPEN, PRINT#, and CLOSE is followed for every sequential file that is written.

General format of sequential files: Figure FMOU-1 shows a listing of a typical INVENT.DO file section.

This portion shows the entries:

```
1ØØØ,12345,32K rams half good,12,24
1Ø1Ø,876554,AK47 children's assault rifle,23,Ø
```

representing two inventory records for a typical Radio Shack store.

Why are there blanks (20H)? There are blanks because commas were used between the PRINT items. The data was written out exactly as it would have been to the display, with "tabs" (see CHTP-100), and those tabs resulted in blanks.

Figure FMOU-1 — *Typical Sequential File*

```
ØØØØ:ØØ = 2Ø 31 3Ø 3Ø 3Ø 2Ø 2Ø 2Ø  2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø    1ØØØ
ØØØØ:1Ø = 2Ø 31 32 33 34 25 2Ø 2Ø  2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø    12345.
ØØØØ:2Ø = 33 32 4B 2Ø 72 61 6D 73  2Ø 68 61 6C 66 2Ø 67 6F   32K rams half go
ØØØØ:3Ø = 6F 64 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø  2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø   od
ØØØØ:4Ø = 2C 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø  2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø    ,
ØØØØ:5Ø = 2Ø 31 32 2Ø 2Ø 2Ø 2Ø 2Ø  2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø    12
ØØØØ:6Ø = 2Ø 32 34 2Ø ØD 2Ø 31 3Ø  31 3Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø    24 . 1Ø1Ø
ØØØØ:7Ø = 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 38 37  36 35 35 34 2Ø 2Ø 2Ø 2Ø        876554
ØØØØ:8Ø = 2Ø 2Ø 2Ø 2Ø 2Ø 41 4B 34  37 2Ø 63 68 69 6C 64 72        AK47 childr
ØØØØ:9Ø = 65 6E 27 73 2Ø 61 73 73  61 75 6C 74 2Ø 72 69 66   en's assault rif
ØØØØ:AØ = 6C 65 2Ø 2Ø 2Ø 2C 2Ø 2Ø  2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø   le   ,
ØØØØ:BØ = 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 32 33  2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø         23
ØØØØ:CØ = 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 3Ø 2Ø  ØD                                Ø .
```

Figure FMOU-2 — *File Using Semicolons*

```
ØØØØ:ØØ = 2Ø 31 3Ø 3Ø 3Ø 2Ø 2Ø 31  32 33 34 35 2Ø 33 32 4B    1ØØØ  12345.32K
ØØØØ:1Ø = 2Ø 72 61 6D 73 2Ø 68 61  6C 66 2Ø 67 6F 6F 64 2C    rams half good,
ØØØØ:2Ø = 2Ø 31 32 2Ø 2Ø 32 34 2Ø  ØD 2Ø 31 3Ø 31 3Ø 2Ø 2Ø    12  24 . 1Ø1Ø
ØØØØ:3Ø = 38 37 36 35 35 34 2Ø 41  4B 34 37 2Ø 63 68 69 6C   876554 AK47 chil
ØØØØ:4Ø = 64 72 65 6E 27 73 2Ø 61  73 73 61 75 6C 74 2Ø 72   dren's assault r
ØØØØ:5Ø = 69 66 6C 65 2C 2Ø 32 33  2Ø 2Ø 3Ø 2Ø ØD            ifle, 23  Ø .
```

Note that there is a comma after each string in a record. This is the only comma appearing in the record and it is in there because we explicitly put it in. Every record is terminated by a **ODH** byte. This is a carriage return, and duplicates the carriage return that would be done for a PRINT line on the display.

Note also that the two records shown here are of different lengths. There are also some other interesting items, such as leading blanks before numeric items.

Obviously some space has been wasted here. Let's try again with semicolons in place of commas. We'll use this PRINT statement:

```
13Ø PRINT#1;PN;MP;DE$;",";NH;NO
```

After the same entries we have the two records shown in Figure FMOU-2.

This time the records occupied much less space. The difference is because the semicolons eliminated the spaces between the items as they were PRINT# ed to RAM. More on sequential file formats later in this procedure.

General flow for reading a sequential file: To read a sequential file, do this:

1. Execute a BASIC OPEN statement. The format of OPEN is

```
1ØØ OPEN"device" FOR INPUT AS file#
```

A typical OPEN might be

```
1ØØ OPEN"RAM:INVENT.DO" FOR INPUT AS 1
```

The above example "opens" a file called INVENT.DO for sequential INPUT from buffer 1.

The OPEN here is similar to the OPEN for creating a file. Here we know the name of the file we wish to OPEN, as it must exist to read it. The buffer used in the OPEN must be a second buffer if a Write File is also taking place. The system file manage software finds the given file name from the RAM directory and locates its area on disk.

The buffer number is the buffer to be used to store file data. The system reads the device in large segments of data, rather than reading each new piece of data, which would be inefficient. Data is accumulated in the buffer until it is filled, and then a read is done from this data.

2. Now that the file is OPENed, you can easily read from it. You may do as many reads as you want. Each read is very similar to an INPUT statement; an INPUT#1 will read from the sequential file in lieu of INPUTting from the keyboard. The number for the INPUT corresponds to the buffer number.

Just as you could specify a list of INPUT items for keyboard input, you can specify a list of INPUT# items. The INPUT# items should correspond to the PRINT# items that were written out to the sequential file. You know this sequence beforehand, of course. Using the inventory example, you should input a part number, manufacturer's part number, description, number on hand, and number ordered, as follows:

```
1ØØ OPEN"RAM:INVENT.DO" FOR INPUT AS 1
11Ø INPUT#1, PN,MP,DE$,NH,NO
12Ø IF EOF(2) THEN GOTO 15Ø
13Ø PRINT PN,MP,DE$,NH,NO
14Ø GOTO 11Ø
15Ø CLOSE 1
```

We've used the same variable names as in creating the file.

3. The EOF statement checks for the "end of file" or EOF. The file manage software knows when this EOF comes up as it reads in the file; it knows the number of records in the file from the disk directory entry describing

the file. At this point there is no more data in the file buffer, as there was in the Write File case. It is good programming practice, however, to CLOSE any buffer to release it to the system for possible other use.

4. This general sequence of OPEN, INPUT#, and CLOSE is followed for every sequential file that is read.

In the example above, the BASIC interpreter would read in an item at a time, looking for the proper delimiter between each data item in the disk file. It would look for spaces between numeric values and for a comma delimiter at the end of strings. For delimit rules, read on further in this procedure.

Method for appending data to an existing sequential file: Suppose that you have a sequential file open with data in it. If you want to add data to it, you cannot simply do an OPEN "device" FOR OUTPUT. . ., because that will OPEN the file for output starting at the first record. Instead, an OPEN"device" FOR APPEND. . . is done, which essentially says, "OPEN the file, find the last record, and get ready to add data at the end". Using our Radio Shack inventory program we'd have

```
100 OPEN"RAM:INVENT.DO" FOR APPEND AS 1
110 INPUT PN,MP,DE$,NH,NO
120 IF PN=-1 THEN GOTO 150
130 PRINT#1,PN;MP;DE$;",";NH;NO
140 GOTO 110
150 CLOSE 1
```

At the end of this session, the INVENT.DO file would have all input data appended at the end.

How to find the end of a file: In the examples above, we used the EOF function to determine where the end of a file was. The general procedure is:

1. OPEN the file
2. Test EOF by IF EOF(n) THEN GOTO XXXX
3. Read the next record by INPUT#
4. Process the data
5. Loop back to 2.

EOF is a handy way to determine the end of a file, but not the only way. You could mark the EOF by writing out a last record with data items set to some unique value, such as -1 for a part number. You might also know beforehand that the number of records will be an exact number. These are perfectly logical ways to determine when the last record has been read, although EOF is a more general case.

Further format considerations: Most of the trouble with sequential files comes from improper delimiting of data items in the records. There's no problem with numeric data items, but there is a problem with strings, because strings may contain some delimiting characters such as spaces or commas. Here are some rules for PRINT#s and INPUT#s:

1. If all of the data items to be written are numeric, there's no problem. The numeric data items will have trailing blanks (ASCII 20H), and they will delimit the data item from the next. Numeric data items, by the way, have either a leading blank or a minus sign, depending upon their value. Of course even here you must have the same number of INPUT# data items as you have data items in the file and in the same order. If you write out variables NH, OR, and DT for each record, you'd better read them back in the same order.

2. A line of output is generated whenever a PRINT# is terminated by an ENTER in place of a comma or semicolon. A line is ended by a carriage return ASCII character (**0DH**) and generally represents one record. You can generate a record of 6 data items by many methods:

```
100 PRINT#1,A;
110 PRINT#1,B;
120 PRINT#1,C;
130 PRINT#1,D;
140 PRINT#1,E;
150 PRINT#1,F          (note no semicolon here)
1000 PRINT#1,A;B;C;
1010 PRINT#1,D;E;F     (note no semicolon here)
2000 PRINT#1,A;B;C;D;E;F
```

The three methods above generate the same record, 6 data items separated by spaces and the last terminated by a carriage return (**0DH**).

Actually, you would never really need a carriage return in the entire file. You could simply write out data items with semicolons. Carriage returns do lend themselves to using the LINEINPUT# command however, which helps in debugging (see below).

3. If you write out strings without quotes around them, a subsequent INPUT# with a string data item will start reading the string with the first non-blank. It will then read in all following characters as part of the string until:

A. A comma is found

B. The end of the file

C. A carriage return (generated by END OF LINE)

D. The 255th character

This poses several problems. If the string you've written has embedded commas, expect to get invalid reads, as the comma will act as a false delimiter. If you've written a string followed by a numeric data item, expect to get that data item and others included as part of the string. One way around this is by the next note.

4. You may enclose strings with double quotes. This involves doing something like

```
100 PRINT#1,CHR$(34);A$;CHR$(34);AS
```

In this example, the CHR$(34) supplies a double quote character in front of and behind the string A$. A$ itself can contain anything — spaces, commas — anything but a double quote that is. The next double quote, carriage return, or end-of-file will delimit the string.

5. If a carriage return (ENTER) is preceded by a down arrow character, the carriage return is not taken as a delimiter, but as part of a string, or it is ignored for a numeric item.

6. Lots of rules, eh? One would think there would be an easier way to do all of this. . . Well. . . . there isn't.

Using PRINT# USING: Use PRINT#n USING in identical fashion as PRINT USING (see PRUU-100). The data will go to the system file n specified in the PRINT# rather than to the screen.

Using LINEINPUT#: LINEINPUT# reads in an entire line from a sequential data file. It stops only when it encounters a carriage return (**0DH**), the end of file, or the 255th character. All of the spaces, commas, double quotes and other anomalies we've been discussing above are included in the line.

LINEINPUT# is handy for "seeing what's out there" and one suspects it was included because everyone was having so darn much trouble with the delimiters in sequential files.

Another good use for LINEINPUT# is in processing ASCII files. You can pretty well read in anything, although some of the non-ASCII data will result in graphics characters or weird screen actions if you try to print it without some processing. LINEINPUT# can be used to read in files, delete unwanted records, and write them back out again. The procedure goes something like this:

```
80 MAXFILES=2
90 CLEAR 1000
100 OPEN"RAM:FILE1.DO" FOR INPUT AS 1
110 OPEN"RAM:FILEB.DO" FOR OUTPUT AS 2
120 IF EOF(1) THEN GOTO 170
130 LINEINPUT#1,A$
140 (process A$)
150 PRINT#2,A$
160 GOTO 120
170 CLOSE 1,2
```

Using multiple buffers: You may OPEN as many files for input and output as you wish. Assign a different buffer number for each file in the OPEN. But read on. . .

The number of buffers available is defined by a BASIC variable called MAXFILES. MAXFILES is set to 1 initially and must be redefined by a

```
MAXFILES=n
```

before you use more than one file for BASIC file operations.

FNMO
Files and File Names, Model 100

The Model 100 treats data as "files". You can look on a file as a collection of similar types of data — all of the names in a mailing list, for example, or all of your creditors (boy, that's a massive file!). These files are "data files".

Another type of file is a "program file". This is generally a BASIC program, although it might be a machine language (see MLWI-100) program as well.

Files may be "routed" to different devices in the system. You might want to store data in RAM in one case and on cassette in another. You might want to "read in" a data or program file from RAM or cassette, or even from the modem as data is received from the telephone line.

Because files can be routed to different devices in the system, these devices are given names:

CAS	Cassette tape
COM	RS-232-C communications
LCD	Screen
LPT	Line printer
MDM	Modem (telephone)
RAM	RAM

Two of these "devices" are "write only" — LCD and LPT. The others are write and read devices.

The advantage of having device names is that data and programs can be easily "routed" to different devices without having to change dozens of BASIC statements or other commands. The OPEN command is used to route files to different devices in BASIC (see FMOU this section). SAVE, LOAD, and other commands also route data or programs to different devices.

When files are used with some of the read/write devices, there is a name associated with the file. Obviously there can't be any name for output to the LCD, LPT, COM, or MDM device, but RAM or cassette tape is capable of storing many different files, and these files must all have a unique name.

COM and MDM devices must have a "configuration" specified along with the COM or MDM prefix. This configuration defines the baud rate, word size, parity, number of stop bits, and XON/XOFF status. See RHSO(100) for a description of these parameters.

Files within the Model 100 are given names for reference. A file name is 1 to 6 characters, the first of which must be alphabetic. In addition to the name, there is a file "extension" of a period followed by two letters. Standard file extensions are:

.DO	Text Document File
.BA	BASIC Program
.CO	Command programs in machine language

The extension is "imposed" by whatever Model 100 program you happen to be in when you create the file. If you're editing a document, for example, the file name will become the 1 to 6 character name you give the file plus ".DO".

There are two standard file names used in the Model 100 — ADRS.DO, used by the the ADDRSS program (ADMO, this section) and TELCOM (ACHT-100 and MHTU-100, this section), and NOTE.DO, used by the SCHEDL program (SCMU, this section).

A complete file name consists of the device designator, a colon, the actual file name, and an extension for RAM and CAS files; the COM/MDM designator, a colon, and a configuration string for RS-232-C or modem files; or a simple LCD: or LPT: for the display or printer. Some of these items are optional depending upon the device and command.

FSMO

Video/Graphics Worksheet, Model 100

Figure FSMO-1 is a full size graphics worksheet for the Model 100. IJG and William Barden, Jr. hereby grant permission to copy it for your own use as many times as is necessary, as long as the number of copies is kept under 250,000.

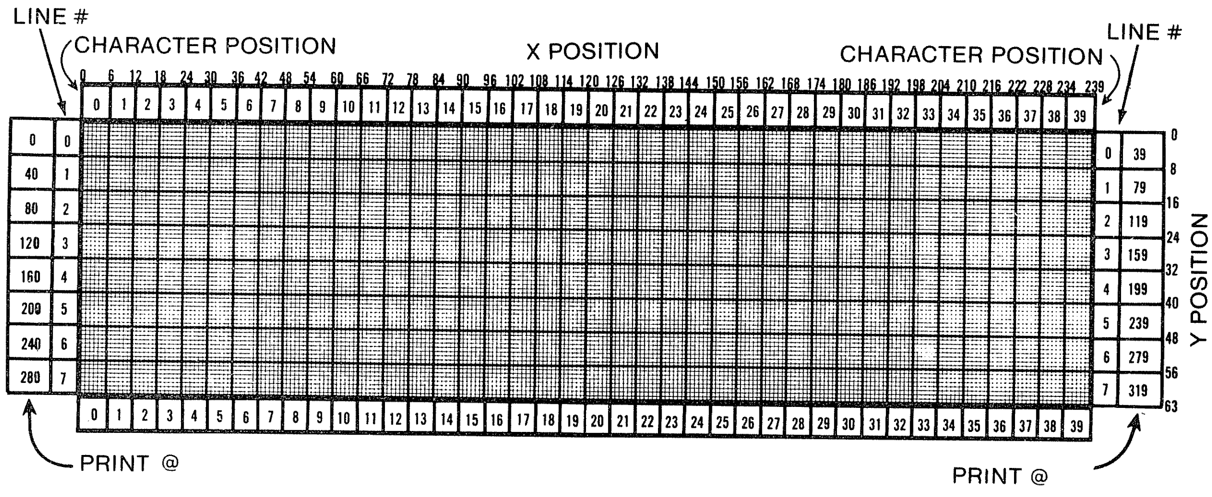


Figure FSMO-1 — *Video Graphics Worksheet, Model 100*

FSWT

Files, Seeing What's There, Model 100

The main menu display displays the system files of

```
BASIC    TEXT    TELCOM    ADDRSS
SCHEDL
```

Files displayed after these are "user" files.

To display the files from within BASIC, use the FILES command. It will display all user files.

Enter

```
FILES
```

You'll see something like:

```
ADRS  .DO    BILL  .DO    NOTE  .DO
DDD   .DO    LPDMP .BA    GGG   .DO
SCRAT.DO
```

FTLO-100

FOR. . .TO Loops, BASIC, Okay to Break Out? Model 100

Read procedure FTLO in the main section. The information there applies exactly to the Model 100 as well.

FTST-100

FOR. . .TO. . .STEP, BASIC, Model 100

The FOR. . .TO. . .STEP command in the Model 100 is identical to the description in FTST in the main section.

GHSO
Graphics, High-Speed, Model 100

There are several efficient methods for high-speed graphics. First, forget about SET/RESET (see PSMO, this section). This is fine for plotting, but terrible for lines, animation, or simple figures.

One easy method to use is the "string method". This method defines each row of graphics characters as a string. As strings can be PRINTed quickly, it is very fast. To use this method:

A. Determine the graphics characters, but not the locations. You'll need to lay out a figure on a graphics worksheet (see FSMO, this section) and translate into the graphics codes and locations.

Suppose that we have a figure of a fish. The figure is shown in GHSO-1.

B. Establish a string for each row of the character. The first row would be a string of the characters 254, 252, and 254. A string can be made of non-text characters by using CHR$ (see CUSE-100). The string for the first row would be defined by

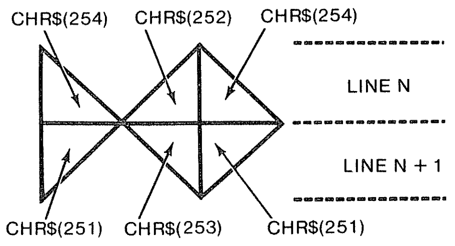
```
100 A$=CHR$(254)+CHR$(252)+CHR$(254)
```

C. Rapidly write out these rows at the proper screen locations by using PRINT @. Typical code would be

```
100 CLS
110 A$=CHR$(254)+CHR$(252)+CHR$(254)
120 B$=CHR$(251)+CHR$(253)+CHR$(251)
130 PRINT @ 220,A$
140 PRINT @ 220+40,B$
150 GOTO 150
```

There are other methods to use for graphics, but the string method is fast and easy to use.

Figure GHSO-1 — *Fish Figure*



notes

HASA-100

Amount of Storage in a BASIC Array, How to Find, Model 100

The Model 100 allows the same variable types as in the Model I, II, and III. In fact, the Model 100 single- and double-precision variables are in bcd or "binary-coded-decimal" format, but the number of bytes allocated per variable is the same. Read the portions of procedure HASA (main section) that apply to the Models I/II/III to find out storage requirements for integer, single- and double-precision, and string arrays.

notes

IDER-100
ID Error, Model 100

See IDER in the main section.

IDFK-100
INPUTting Data From the Keyboard, BASIC, Model 100

Read procedure IDFK in the main section. Information on INPUT applies equally well to the Model 100.

INAR-100
Initializing Arrays, BASIC, Model 100

Read procedure INAR; it applies to the Model 100 as well.

INHU-100
INKEY$, How to Use, BASIC, Model 100

All of the material under INHU applies as well to the Model 100. In addition, pressing an undefined Function Key, PASTE, or LABEL will return an ASCII 0 (*not a null!*) with a length of 1 character.

IOHT-100
INP, OUT in BASIC, How to Use, Model 100

The information about INP and OUT in procedure IOHT, main section, applies to the Model 100 as well. The microprocessor involved here is an 80C85, which has an IN and OUT machine language instruction similar to the Z-80, accomplishing the same thing, reading or writing 8 bits of data to an "I/O port". At this time of writing, the Model 100 I/O port addresses have not been made public.

IPNF-100
Integer Portion of a Number, Finding, BASIC, Model 100

Read procedure IPNF in the main section. Information there applies exactly to the Model 100.

ITEL-100
IF. . .THEN. . .ELSE, BASIC, Model 100

Read procedure ITEL in the main section; it applies equally well to the Model 100.

IVHU-100
Integer Variables in BASIC, How to Use, Model 100

Integer variable use in the Model 100 is the same as used on the Models I, II, and III. Read procedure IVHU in the main section.

IWAT
Interrupts, What Are They?, Model 100

The Model 100 has a number of **interrupts**. An interrupt is an external event which interrupts a program that is executing, performs a special action, and then lets the program resume its processing. (In some cases, the interrupt is rather catastrophic, and the program never resumes. An example might be the case of an external interrupt to a nuclear power plant control computer that interrupts a parts list printout to inform the main computer that core melt down is occurring. If the programmer is on his toes, he might take special program actions, rather than resuming the printout. . .)

Why use interrupts? In the general case, mainly to enable the computer to perform a "background" task while also servicing an infrequent interrupt. This increases the overall efficiency of a computer system and permits more to be accomplished than devoting the entire computer system to waiting for that "foreground" action which may occur only several times per second.

In the Model 100, interrupts are used mainly to signal interesting "external" events that cause a special action, and there's not really that "foreground/background" processing described above — but the interrupts are still powerful.

In the Model 100 interrupts can be setup for a specific time of day (see TIMO, this section), a BASIC program error condition (see ETIB, main section), a Function Key press (see FKMO, this section), a modem character coming in (see MIMO, this section), or an RS-232-C character coming in (see MIMO, this section).

Believe it or not, the interrupt structure is almost like that of a "big" computer. The interrupt causes a branch to a special set of BASIC instructions defined by the user. This is the "interrupt processing subroutine" and is defined by an ON TIME$ GOSUB XXXX or similar command in the BASIC program, where XXXX is a BASIC line number. The interrupt can be turned ON and OFF by a TIME$ ON or TIME$ OFF or similar command, so that the interrupt either causes an interrupt action or not. Lastly, the interrupt can be turned off, *but still remembered* by a TIME$ STOP or similar command; a subsequent TIME$ ON then immediately causes the special interrupt action if the interrupt has occurred during the time in which it was "stopped". This is darn sophisticated, and almost makes up for the feeble Schedule Organizer/Address Organizer search function!

The STOP command lets you turn off one interrupt while processing another "higher-level" interrupt. If the "lower-level" interrupt than occurs, it is still "remembered" and can be processed after the higher-level interrupt has been handled! Kudos to the Shack.

LBAP-100

Listing BASIC Programs, Model 100

Use the LIST command to list the BASIC lines on the display, or the LLIST to list on the system line printer. LIST nnn,mmm as in LIST 100-300 lists all specified lines. Use the minus sign (–) to list all lines up through or from a specified line, as in LIST–300 (beginning line through 300) or LIST 300– (line 300 through end line).

See TSPD-100 for notes on stopping the display.

LBWI

Low Battery Indicator, What is It?, Model 100

The Shack says you have about 15 minutes of power left, although I've found it to be much more than that. Change the 4 AA batteries, or plug the Model 100 into the ac adapter. Don't worry about losing memory files or data when changing batteries, by the way, or about changing the batteries in 15 minutes. It's extremely difficult to lose any data in the Model 100 as the Ni-Cad (nickel cadmium) batteries will retain memory data for 8 to 30 days. Once the AA batteries go, however, you won't be able to compute, but the memory will still be there (at least for a week or so)!

LBWT-100

LET, In BASIC, What to Do About It, Model 100

Forget about it. LET was put into the Model 100 BASIC to make the BASIC compatible with earlier BASIC versions that required the LET, as in "100 LET A=1.234". None of the Radio Shack BASICs require it, and you can simply use "100 A=1.234".

LFBA-100

Line Format, BASIC, Model 100

Read procedure LFBA in the main section. Model 100 BASIC lines are stored in the same format. The pointers and line numbers are stored in "8085 address format", which is identical to "Z-80 address format", least significant byte followed by most significant byte (see EFAF, this section).

LLBO

LABEL Line, BASIC, Getting Rid of, Model 100

To get rid of a possible label line on the display in BASIC, do

```
SCREEN Ø,Ø
```

This makes the last line on the screen available for scrolling while executing BASIC programs.

```
SCREEN Ø,1
```

resets the labels and allows scrolling only on the first 7 lines.

This command can be used in BASIC to display or kill the label line at will.

LMCO

Loading Machine Language Files from Cassette, Model 100

Machine language (see MLWI, this section) files are applications programs such as word-processing programs that are not written in BASIC or such things as memory DUMPs that you have created by using the SAVEM command in BASIC (see SMLO, this section).

Enter BASIC (see BHTO, this section).

If the program is to be used in conjuction with BASIC programs, protect the portion of RAM memory into which you want to load the applications program by a

```
CLEAR sss,mmmmm
```

where sss is a string storage area amount (typically 100 or 200), and mmmmm is **one less** than the first location of the machine language program (CLEAR 100,49999 protects RAM locations 50000 on up).

Now load the application cassette into your cassette recorder. Connect the plugs as shown in CRPI-100. Rewind the cassette to the beginning or to the point directly before the file you wish to load, if you can locate it fairly exactly.

You now have two choices.

If you want to load without executing the file, as in simply loading a data file into memory, do

```
LOADM "CAS:NAME"
```

where "NAME" is the 1- to 6-character name of the cassette file you want to load. If you don't know the name, or want to disregard it, simply do a

```
LOADM "CAS:"
```

Another shorter form of this command is CLOADM, which automatically loads a machine language file from cassette:

```
CLOADM "NAME"
```

If you want to load and execute, as in loading a machine language applications program, do

```
RUNM "CAS:NAME"
```

or

```
RUNM "CAS:"
```

where name is the 1- to 6-character name of the cassette file you want to load.

Up to this point, there should have been no tape motion. When the command is completed, you should see the tape move as the data is read in. At the same time, you should hear a high-pitched sound as the tape is read. You'll also see any file name that comes before the file you're looking for displayed as

```
Skip: XXX
```

where XXX is the skipped file name. When the file you've specified is found, you'll see:

```
Found:XXX
```

displayed on the screen, and the file will have been loaded. After the file has been loaded, you'll see

```
Top: XXXXX
End: XXXXX
Exe: XXXXX
```

indicating the start, end, and execution address for the file.

If a RUNM command has been used, execution of the program will start immediately.

LMFO

LOADing Multiple Machine Language Files from Cassette, Model 100

No problem. Simply execute one CLOADM (see LMCO, this section) after another. The files will load as specified in the load addresses in the file. As long as you CSAVEMed the files with non-overlapping addresses (see SMLO, this section), the files should not conflict with one another.

LMRO-100

Loading Machine Language Files from RAM, Model 100

Machine language (see MLWI) files are applications programs such as word-processing programs that are not written in BASIC or such things as memory DUMPs that you have created by using the SAVEM command in BASIC (see SMLO, this section).

Enter BASIC (see BHTO, this section).

You now have two choices.

If you want to load without executing the file, as in simply loading a data file into memory, do

```
LOADM "RAM:NAME"
```

where "NAME" is the 1- to 6-character name of the cassette file you want to load. You don't need the "RAM" portion of the command and you can simply do a

```
LOADM "NAME:"
```

If you want to load and execute, as in loading a machine language applications program, do

```
RUNM "RAM:NAME"
```

or

```
RUNM "NAME"
```

where name is the 1- to 6-character name of the cassette file you want to load.

After the file has been loaded, you'll see

```
Top: XXXXX
End: XXXXX
Exe: XXXXX
```

indicating the start, end, and execution address for the file.

If a RUNM command has been used, execution of the program will start immediately.

LOHT-100

Logical Operations in BASIC, Model 100

All of the information in LOHT, main section, applies equally well to the Model 100. Also note that the three Model II operators of XOR, EQV, and IMP are implemented on the Model 100 as well, so read all of procedure LOHT.

LSBD

Loading and Saving a BASIC Program to a Device, Model 100

Loading and saving a BASIC program to a system device such as RAM, CAS, COM, or MDM uses the BASIC commands LOAD and SAVE. These commands operate similarly to CLOAD and CSAVE for cassette tape. As a matter of fact,

```
LOAD"CAS:filename"
```

is identical to

```
CLOAD"filename"
```

and

```
SAVE"CAS:filename"
```

is identical to

```
CSAVE"filename"
```

for cassette operations (see SBPT-100).

The purpose of LOAD and SAVE is to enable you to read in a BASIC file from **any** device and to write out to **any** device. This is a very handy command for such things as "downloading" BASIC files from a data communications link.

Generally, you'll be LOADing or SAVEing a BASIC file from or to RAM and you'll have:

```
LOAD"RAM:filename"
SAVE"RAM:filename"
```

These commands simply load in the specified BASIC file, which can then be executed or write out an existing program in identical fashion to CLOAD and CSAVE.

To execute the BASIC program directly after LOADing, do

```
LOAD"RAM:filename",R
```

Another way of doing this is to use the RUN command:

```
RUN"RAM:filename",R
```

Here, the optional R will keep all opened files open; this is handy for maintaining "common" files for more than one BASIC program or program segment.

To save a BASIC file in ASCII (see AFWA-100), use the "A" option with CSAVE

```
CSAVE"RAM:filename",A
```

To LOAD or SAVE using the communications line or modem, use the device prefix COM or MDM. In these cases, you'll have to specify the "configuration" for the baud rate, word length, parity, stop bits, and XON/XOFF as descibed in RHSO(100).

```
LOAD"MDM:701E"
```

loads in a BASIC file from the modem with 7-bit words, odd parity, one stop bit, and XON/XOFF enable.

LSER-100
LS Error, Model 100

See LSER in main section.

notes

MAFO
Merging BASIC Files, Model 100

To MERGE two BASIC files, get the first file into memory by loading from RAM, CASsette, COM, or MDM (or you may have entered the file from scratch). The secret for the MERGE is that the second file to be merged must be an ASCII file (see AFWA-100). Use the "A" option for CSAVE or SAVE to write out the second file in ASCII prior to the merge. Now use the

```
MERGE"device:filename"
```

command to read in the second file. The device and filename can be any Model 100 device and filename (or configuration for COM/MDM files). See FNMO(100).

The second file will now be loaded and its BASIC lines *merged* with the existing BASIC lines. This means that if there are identical line numbers, the BASIC line from the second file will replace the first BASIC line, so be careful. Best to do the MERGE with two BASIC programs whose line numbers are "non-conflicting". The first BASIC program might have line numbers from 100 through 900, while the second might have line numbers from 2000 through 2900. The resulting BASIC program will have line numbers of 100 through 900, followed by line numbers of 2000 through 2900.

MFAL-100
Memory, Finding Amount Left, Model 100

In BASIC command mode, do a PRINT FRE(0) to find the number of bytes left in the user RAM. FRE(0) can also be used inside a BASIC program to dynamically check on the memory remaining. The expression inside the parentheses is a "dummy" expression, and has no effect on the computation, with this exception: If the expression is numeric, then the amount of RAM free space is returned; however, if the expression is a string, then the amount of string free space is returned. PRINT FRE("") or PRINT FRE(A$) prints the amount of free string space (see SSFA-100).

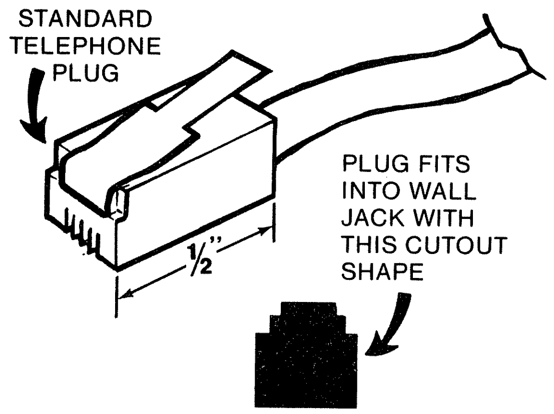
MHTU-100
Modem, How to Use, Model 100

Read procedure MWAT if you know nothing about modems.

Also read procedure RHSO(100) to make certain that your communications parameters are set up properly for the data communications system you'll be calling.

If you cannot directly connect to a telephone line and have an acoustic coupler (Radio Shack 26-3805), see procedure ACHT-100. If your telephone has a standard telephone type jack (see Figure MHTU-100-1), you can directly connect to the telephone line, providing you have the Model 100 Modem Cable (RS 26-1410).

Figure MHTU-100-1 — *Telephone Jack*



Follow these steps to use the modem on the Model 100:

1. Set the DIR/ACP switch on the left-hand side of the Model 100 to DIR, or Direct Connect.

2. Plug the "DIN" connector (the larger round connector) on the modem cable into the PHONE connector on the rear of the Model 100.

3. There are two leads coming out of the modem cable, one silver, and one beige. Connect the leads as shown in Figure MHTU-100-2. The beige cable goes into the wall jack, while the silver cable connects the phone. The "shorting plug" is used when the Model 100 is to be disconnected from the whole scheme. Frankly, I'd simply reconnect the telephone to the wall jack rather than using the shorting plug.

4. You're now ready to dial up a data communications system such as CompuServe or a bulletin board. There are a number of ways to do this, dependent upon whether you have a number stored in a RAM file, whether you want to keyboard in a number, or whether you want to dial the number manually. We'll start from the simplest first:

5. **Dialing a number manually**:

A. If you are originating the call, set the modem switch for ANS/ORIG on the left-hand side of the Model 100 to "Originate".

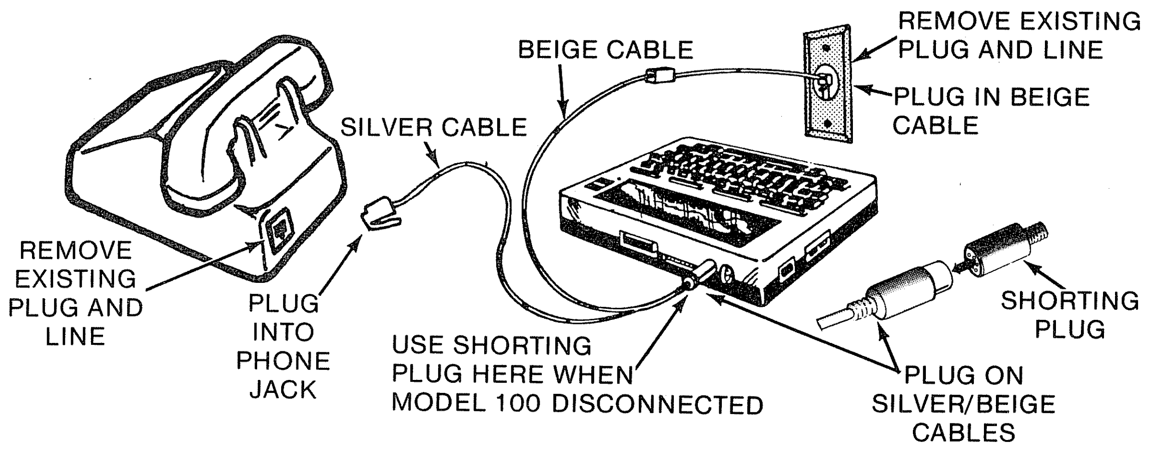
B. Enter TELCOM by procedure TEHT(100).

C. Dial up the number of the Bulletin Board or network while listening on the phone. After the phone is answered, you'll hear a pause followed by a high-pitched whine of the "carrier" frequency. Take your time (you have a minute or so), and continue.

D. Press Function Key 4 (TERM) to enter "terminal mode". You should see a new set of labels above the Function Key indicators:

```
Prev Down Up  Full              Bye
```

Figure MHTU-100-2 — *Model 100 Modem Cable*


REMOVE EXISTING PLUG AND LINE
BEIGE CABLE
PLUG IN BEIGE CABLE
SILVER CABLE
REMOVE EXISTING PLUG AND LINE
PLUG INTO PHONE JACK
USE SHORTING PLUG HERE WHEN MODEL 100 DISCONNECTED
SHORTING PLUG
PLUG ON SILVER/BEIGE CABLES

E. Press Function Key 4 (FULL/HALF) depending upon whether the communications system uses "full" or "half duplex". Pressing Function Key 4 will "toggle" the setting between full and half duplex. Most systems will use full duplex, so use that option if you aren't certain.

F. You should now see the prompt message of the Bulletin Board or network on the screen. If you do not, try typing ENTER a few times. If you still have trouble, review the steps above and try again with another system (preferably). If you do see data on the screen, but it's garbled, go to procedure RHSO(100). If you see meaningful data, continue with the procedure for Bulletin Boards (BBUS-100) or CompuServe (CPSU-100).

6. **Dialing a number from a RAM file:**

A. The automatic dialing feature works from the ADRS.DO file. See procedure ADMO(100) on how to create this file. The ADRS.DO file generally holds any number of names, addresses, and telephone numbers. Telephone numbers are "bracketed" by colons. The telephone number for auto dialing must also have a trailing <> as in

```
CPS :5551212<>
```

The "<>" causes entry into the Terminal Mode after the dialing. You can have any number of these types of numbers in the ADRS.DO file, and you can get to the proper one by the Find function of TELCOM.

B. If you are originating the call, set the modem switch for ANS/ORIG on the left-hand side of the Model 100 to "Originate".

C. Enter TELCOM by procedure TEHT(100).

D. Press Function Key 1 (FIND). Now enter either nothing, if you have only one number in the ADRS.DO file, or the name or identifier of the communications service you want to call. If ADRS.DO consisted of

```
CPS :5558Ø6Ø<>
Source :5551289<>
Model 1ØØ Net :5559871<>
```

for example, you might enter 100 as the "Find" string. TELCOM would search the ADRS.DO file, find the Model 100 Net number and display it:

```
Telcom: Find 1ØØ
Model 1ØØ Net :5559871<>
```

Make certain the number for the call is in this format and displayed before continuing.

E. Press Function Key F2 (CALL). You'll see

```
Calling Model 1ØØ Net :5559871
```

F. After the number is dialed you'll hear the ring. When the number is answered, you'll see a switch to "Terminal Mode" as indicated by a new set of labels above the Function Key indicators:

```
Prev Døwn  Up  Full                    Bye
```

G. Press Function Key 4 (FULL/HALF) depending upon whether the communications system uses "full" or "half duplex". Pressing Function Key 4 will "toggle" the setting between full and half duplex. Most systems will use full duplex, so use that option if you aren't certain.

H. You should now see the prompt message of the Bulletin Board or network on the screen. If you do not, try typing ENTER a few times. If you still have trouble, review the steps above, and try again with another system (preferably). If you do see data on the screen, but it's garbled, go to procedure RHSO(100). If you see meaningful data, continue with the procedure for Bulletin Boards (BBUS-100) or CompuServe (CPSU-100).

MIMO
Modem Interrupt, BASIC, Model 100

The Model 100 contains a modem interrupt, along with an interrupt for time and Function Keys. If this interrupt is enabled, incoming data to the modem (see MWAT/100) will cause an automatic transfer to a specified "modem interrupt subroutine". This interrupt enables such functions as dumping data via the telephone lines until interrupted by a remote keypress, decoding the keypress in the modem interrupt processing routine, dumping another set of data until interrupted, and so forth.

To GOSUB a modem interrupt processing subroutine at line 10000, for example, you'd have this line at the beginning of your program:

```
1ØØ ON MDM GOSUB 1ØØØØ
```

If you then did an MDM ON command, BASIC would enter the subroutine at line 10000 after it received the next modem character. Because a GOSUB is executed, you can always get back to the "interrupted point" by doing a RETURN at the end of the 10000 subroutine.

What's in the 10000 subroutine? Whatever you want — but usually code related to interpreting that remote data.

To disable the ON MDM interrupt, do

```
MDM OFF
```

To enable it again, do

```
MDM ON
```

To disable the ON MDM interrupt, but cause the Model 100 to remember the interrupt occurred, do

```
MDM STOP
```

When the interrupt is again enable by MDM ON, the MDM STOP command causes an immediate GOSUB action to the ON MDM subroutine. Suppose that you had the modem processing subroutine defined by an ON MDM GOSUB 10000, but disabled it by MDM OFF just before a character came in via the modem. If you then enabled it by MDM ON, nothing would happen. However, if you had disabled it by MDM STOP before the character came in and again enabled it by MDM ON after the character came in, an immediate GOSUB 10000 would occur after the MDM ON.

ON COM GOSUB, COM ON, COM OFF, and COM STOP work exactly the same as the modem commands, except that data would be coming in via the RS-232-C port rather than through the modem, an audio input.

MLIM
Memory Left, in Model 100

The amount of free memory in the Model 100 is indicated on the main menu display in the lower-right hand corner. This figure is adjusted with each user file added or deleted from memory. One text character takes approximately one byte.

Whenever you enter BASIC, the number of free bytes are displayed after the BASIC title message

```
TRS-80 Model 100 Software
Copr. 1983 Microsoft
15409 Bytes free
OK
```

This number will be smaller than the number on the main menu, as BASIC utilizes part of free memory for "working storage".

MLWI-100
Machine Language, What Is It?, Model 100

Read procedure MLWI in the main section. Although there is no officially released assembly program at this time of writing, the Model 100 is capable of being programmed in both machine and assembly language in 80C85 code. This code is a rudimentary language for the microprocessor used in the Model 100. The 80C85 microprocessor is roughly comparable to the Z-80 (Models I/II/III) and 6809E (Color Computer), albeit somewhat slower because it is a low-power "CMOS" (complementary metal-oxide semiconductor) design. The 80C85 instruction set is identical to the Intel 8085 microprocessor.

MMGB
Main Menu, Getting Back to, Model 100

Function Key F8 normally takes you back to the main menu from applications programs. One exception: The Terminal program of TELCOM (MHTU-100) must be terminated by Function Key 8 (BYE) plus one more Function Key 8 keypress to get back to the main menu.

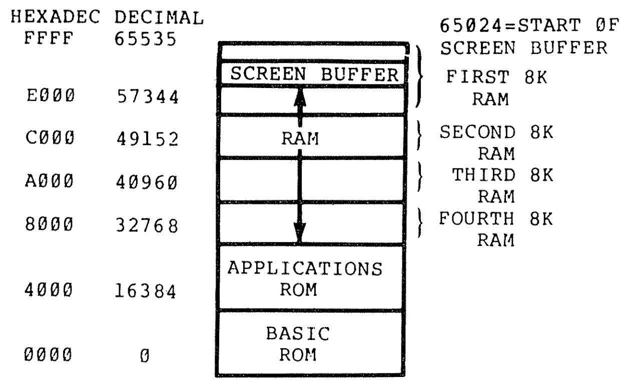
To get back to the main menu from BASIC: Enter MENU alone in the command mode. To get back to the menu at the end of a BASIC program, also use the MENU command.

```
1900 MENU        'return to menu
```

MMMO
Memory Map, Model 100

Figure MMMO-1 shows a memory map of the Model 100.

Figure MMMO-1 — *Memory Map, Model 100*



MOBO
MOD Operator in BASIC, Model 100

The Model 100 MOD operator allows you to find the "modulus" result. What's a modulus? Glad you asked. . .

Do a divide of two numbers. The remainder is the modulus result. Example: 102/9=11, remainder 3. The value 3 is the "100 MOD 9" result. Not too handy, you say? Well, not that uncommon. One example: If a timer starts counting in minutes, the "minute hand" position on a clock can be given by

```
Elapsed Time in Minutes MOD 6Ø
```

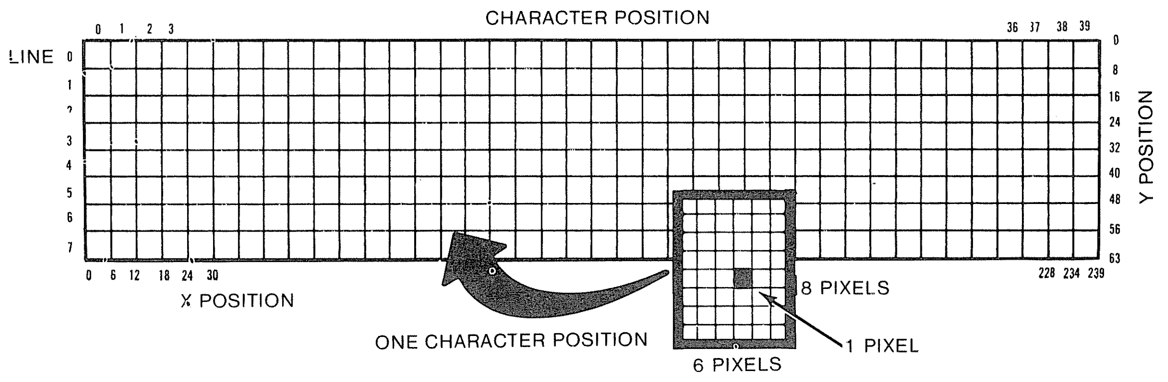
MOER-100
MO Error, Model 100

See MOER in main section.

MOPA
Model 100 Pixel Addressing

All graphics commands in Model 100 BASIC use a high-resolution mode to specify the x,y coordinates. The x coordinate is the horizontal coordinate, while the y coordinate is the vertical coordinate. In this mode (240 by 64), x may be 0 through 239, and y may be 0 through 63. These values define the smallest graphics element that may be displayed, one pixel, or PICture ELement. (Each character position is 6 pixels by 8 pixels). See Figure MOPA-1.

Figure MOPA-1 — *Model 100 Screen Character Positions*



MSLH-100
Multiple Statement Lines in BASIC, How to Use, All Systems, Model 100

You can easily combine several statements in one BASIC line. Read MSLH in the main section to see how.

MTOO
Motor, Turning On in BASIC, Model 100

In BASIC, simply execute MOTOR ON or MOTOR OFF to turn the cassette on or off. You can use this command to control external devices by connecting the two pins that would normally go to the REMote jack on the cassette recorder as described in CCAM-100.

MWAT-100
Modems, What Are They?, Model 100

Read procedure MWAT in the main section. All of the information there applies directly to the Model 100. The Model 100 has a built-in modem that provides an "auto-dial" capability; you can dial a number automatically from the keyboard. Note that this dialing is a "pulse" type dialing, rather than a tone dialing, presumably because "CMOS" integrated circuits incorporating tone dialing were not available at the time of the Model 100 design. To set the pulse rate, see RHSO(100); standard is 10 pulses per second.

See ACHT-100 for information on "acoustic couplers".

notes

NFER-100

NF Error, Model 100

See NFER in main section.

NRER-100

NR Error, Model 100

See NRER in main section.

notes

ODER-100
OD Error, Model 100

See ODER in main section.

OGHU-100
ON. . .GOSUB, ON. . .GOTO, BASIC, Model 100

All of the information in OGHU applies equally well to the Model 100.

OMER-100
OM Error, Model 100

See OMER in main section.

OSER-100
OS Error, Model 100

You haven't defined an area that BASIC can use for string manipulations, or you haven't defined a large enough area. Use a CLEAR command at the start of your program to define a larger string "working storage" area. This would be a command similar to "100 CLEAR 2000,49999"; in this case 2000 bytes are set aside, and the "protected area" is set to location 50000.

How much storage is required for strings? You could go through and laboriously calculate this, but no one does. If your BASIC program is small and you have 8K, set aside 200 or 300 bytes. If you have a large BASIC program, you'll have to trade off string working storage space with space used for the program, space used for variable storage, and so forth; see memory map at MMMO(100). If you have a very large program, you may have to reduce your program size to fit everything into memory.

OVER-100
OV Error, Model 100

See OVER in main section.

notes

PHTU-100
Parentheses, How to Use, Model 100

The information in PHTU applies exactly to the Model 100.

POOM
Power Off and On, Model 100

The Model 100 will turn itself off after about 10 minutes from the last keypress to save batteries. To change this time delay, enter BASIC (see BHTO, this section) and do a

```
POWER n
```

where n is 10 to 255 in 1/10th minute increments. To change the delay to 1 minute (recommended for people who know *exactly* what they are doing) do

```
POWER 1Ø
```

Once changed, this delay will remain in force until a new POWER command overrides it (or until memory power is turned off).

```
POWER CONT
```

disables the power off option; power is never turned off. This should probably be used only if you have the ac adapter continuously connected.

You can also turn power off from inside a BASIC program:

```
1Ø POWER OFF          'program done, turn off power
```

Another form of this is

```
1Ø POWER, RESUME
```

In the first case turning power back on will bring you back to the main menu; in the latter case, the next BASIC statement will be executed.

PPKU-100
PEEK and POKE, Using, Model 100

Read procedure PPKU in the main section. PEEK and POKE are used exactly the same as on other systems, with the exception that you do not have to use the "XXXXX-65536" form of the address for PEEKing or POKEing into locations above decimal 32767. For a memory map of the Model 100 see procedure MMMO.

PROT-100
Protecting Memory in BASIC, Model 100

To protect memory:

On powering up or at any time: CLEAR XXX,MMMMM. The area above MMMMM will not be used. The value XXX is the number of bytes that will be allocated as "string storage area" and is typically 200 bytes; allocate more if you will be doing a great deal of string processing. The area above MMMMM can now be used for any purpose without the BASIC interpreter and other system programs using it for system functions. CLEAR 200,49999, for example, allocates 200 bytes for string storage and protects the area from 50000 to MAXRAM (see RFMO, this section).

Protected memory is normally used for storage of user machine language programs (see MLWI-100), but can also be used for special data areas.

See procedure MMMO(100) for the memory map of the Model 100.

PRUU-100
PRINT USING, BASIC, Using, Model 100

PRINT USING in the Model 100 is used almost the same as in other systems. Read PRUU in the main section. Carets are used for exponential format in place of up arrows. Carets are entered by pressing SHIFT followed by 6. Backslashes are used in place of "%" characters for generating string data. Backslash is entered by pressing GRPH followed by minus.

PSMO
PSET, PRESET in BASIC, Model 100

PSET and PRESET set or reset a graphics point (see MOPA(100).

The format for setting a POINT is

```
PSET (x,y)
```

where x and y are the coordinates of the point.

The format for resetting a point is

```
PRESET (x,y)
```

where x and y are the coordinates.

PTSC-100
Printing the Screen, Model 100

To print the screen contents (in most instances), press the PRINT key (upper row). The contents will be "dumped" to the system line printer.

PVMO-100
Passing Variables to Machine Language Programs, Model 100

If you are interfacing BASIC to a machine language program or programs (see CMLO, this section), you may want to pass "parameters" back and forth. An example of this is a machine language subroutine that takes the time and number of the day of the year and converts it into elapsed seconds. The parameters passed **to** the subroutine would be current time and the "Julian" day, the number of the day of the year from 1 to 366. The parameter passed **back** would be the total number of elapsed seconds.

How can the parameters be passed?

If no parameters are to be passed, then the CALL command simply calls the machine language subroutine without any special data in the A and HL registers.

```
100 A=CALL 60000
```

The machine language subroutine will be called and executed, but no data will be passed between BASIC and the machine language program.

If one or two parameters are to be sent **to** the subroutine, but no parameter is to be returned, then one or two arguments can be included after the address parameter in the CALL command. The first of these is the value to be put into the A register, while the second is the value to be put into the HL register. The first argument must be from 0 through 255, while the second may be –32768 through 65,535. Suppose we wanted to pass a value of 247 to the machine language subroutine at location 60000. We'd have something like

```
100 A=247
110 CALL 60000,A
```

If we wanted to pass two arguments, say a value of 247 and a 16-bit value of 10000, we'd have something like this:

```
100 A=247
110 B=10000
120 CALL 60000,A,B
```

Of course, we could simply have numeric constants, too:

```
100 CALL 60000,247,10000
```

What about **multiple arguments**? The USR call mechanism only allows one 8-bit and one 16-bit integer argument to and from the subroutine. If more then one argument is to be passed then we've got to get clever. Pack two 8-bit arguments by code such as

```
100 A=123*256+45.  'first arg=123, second=45.
110 B=247          'third arg=247
120 CALL 60000,B,A
```

This can be extended into six 4-bit arguments, and so forth.

If you have many arguments to be passed, or if you want to pass arguments **back** from a subroutine, then you must go to an **argument list** concept. In this method the arguments to be passed to and from the subroutine are either in a predefined area of memory or a **pointer** to an argument list is passed. Suppose that an X, Y, and Z value were to be passed to a subroutine. We could define the argument area at locations 57344, 57345, and 57346; POKE them before calling the subroutine; and then do the CALL. Alternatively, we could POKE the arguments somewhere else in RAM and then pass a 16-bit pointer to them by the CALL, as in

```
100 POKE 57344,X    'X to E000H
110 POKE 57345,Y    'Y to E001H
120 POKE 57346,Z    'Z to E002H
130 CALL 60000,0,57344 'call SUBR 1st set
140 POKE 57347,X    'X to E003H
150 POKE 57348,Y    'Y to E004H
160 POKE 57349,Z    'Z to E005H
170 CALL 60000,0,57347 'call SUBR 2nd set
```

Arguments can also be POKEd into dummy strings and arrays, but this can get somewhat messy, as string locations and arrays are **dynamically** moved in RAM as BASIC code is edited, new variables are added, etc. Use the VARPTR function carefully if you take this tack, and make certain that the VARPTR directly precedes the USR call without introducing a new variable name.

QWII-100

Quote, in BASIC, What Is It?, Model 100

Read QWII; the information there applies to the Model 100 as well.

notes

RBWI-100
Reset Button, Where Is It?, Model 100

Model 100: Rear of unit offset from center towards the right, marked "RESET". Pressing RESET returns you to the main menu for the Model 100.

RCCO
RS-232-C Connector, Model 100

The Model 100 uses a standard "DB-25" female RS-232-C connector. The signals on this pin are standard signals for all Radio Shack computers that use this connector. For information on RS-232-C signals see RSWI. The RS-232-C connector is located in the back center of the Model 100 (see SCMO, this section).

RCWA-100
ROM Calls, What Are They?, Model 100

Read procedure RCWA in the main section. The information there is correct; at this time of writing, however, Radio Shack has not released the locations of ROM subroutines. This information will be forthcoming.

RDHT-100
Random Data, How to Generate, Model 100

Read procedure RDHT in the main section for background if you know nothing about random numbers. Then return here.

RND in the Model 100 works similarly to RND on the other Radio Shack systems except that it generates a random number between 0 and 1, a fractional number. The first three random numbers generated by

```
1ØØ FOR I=Ø TO 2
11Ø PRINT RND(1)
12Ø NEXT I
```

are:

```
.59521943994623
.1Ø65862 8Ø5Ø158
.76597651772823
```

It's easy enough to convert these to numbers greater than 1 — simply multiply by the highest number you'd like to have. If, for example, you'd like random numbers from 0 through 9999, multiply by 1000:

```
1ØØ FOR I=Ø TO 2
11Ø PRINT RND(1)*1ØØØ
12Ø NEXT I
```

and you'd get 595.21. . ., 106.58. . ., and 765.97. . .

It's also easy to find **integer** numbers:

```
1ØØ FOR I=Ø TO 2
11Ø PRINT INT(RND(1)*1ØØØ)
12Ø NEXT I
```

You'd get 595, 106, and 765 in this case.

If the numeric expression inside the RND parentheses is 0, the last random number generated would be returned.

```
1ØØ PRINT RND(Ø)
11Ø GOTO 1ØØ
```

would continually print the last random number generated.

Want to reseed the random number generator, as described in RDHT? You *can* invoke RND a random number of times by using the current number of seconds, as in the Model 100 manual (use SEC= VAL(RIGHT$(TIME$,2)), but this is cumbersome. A better way is to do a POKE of seconds into the random number generator seed value. The location of this BASIC "working variable" is not known at this time of writing.

RERF
Renaming a RAM File, Model 100

To rename a RAM file, go to BASIC (see BHTO, this section) and enter

```
NAME "oldfile" AS "newfile"
```

where oldfile is the old file name and newfile is the newfile name. You can also use the form

```
NAME "RAM:oldfile" AS "RAM:newfile"
```

if you like typing.

Both the oldfile and newfile names must be legitimate Model 100 file names (see FNMO, this section). You can't change extensions when you rename. The newfile name can't exist, of course. You can also specify a string variable as either the newfile or oldfile. In this case, the string variable(s) must be legitimate file names, complete with extensions.

RFDO
Ram File, Deleting, Model 100

Enter BASIC (see BHTO, this section).

Use the bellicose KILL command. The format is:

```
KILL "name"
```

where name is a standard filename (see FNMO, this section) with extension, if any. If the extension is not used, the file name will not be found, and the KILL will not be successful.

If you have insufficient free memory, you may not be able to KILL the file. In this case, delete BASIC program lines by typing the line number. Another option: Go to TEXT, select a file, and put it into the PASTE buffer to free up memory space; then return to BASIC to KILL the desired files. Then go back to TEXT and COPY the file back to the original area. See TUHO (100)for descriptions of these actions.

RFMO

RAM, Finding MAX, Model 100

The MAXRAM function returns the "last" RAM memory location available for your machine. This value can't be redefined.

```
CLEAR nnn,MAXRAM
```

allocates nnn bytes for string storage (see OSER-100) and resets any protected area to "MAXRAM".

What's the difference between HIMEM and MAXRAM? HIMEM is the current last RAM location available to BASIC. If you've protected part of memory by a CLEAR command, HIMEM will be set to the value of the CLEAR.

```
CLEAR 1ØØ,59999
```

for example, clears 100 bytes for string working storage and protects memory from 60000 on. HIMEM is 59999, the highest memory location available to BASIC, while MAXRAM is still 62960, the greatest RAM memory location available in the system.

RGER-100

RG Error, Model 100

See RGER in main section

RHSO

RS-232-C Interface, How to Set, Model 100

Read RSWI if you are unfamiliar with data communications.

The RS-232-C interface is set completely under program control; there are no switches to set manually, as in other computers, such as the Radio Shack Model I. If you are using a precanned program, then the program might setup the RS-232-C for you; refer to the documentation in the program for connection information. If you are not using a precanned program and want to define the "communications protocol", continue.

Current communications parameters: The current communications parameters are displayed on entry to TELCOM. See TEHT(100) for information on how to enter TELCOM. After entry you'll see

```
M7I1E,1Ø pps
Telcom:
```

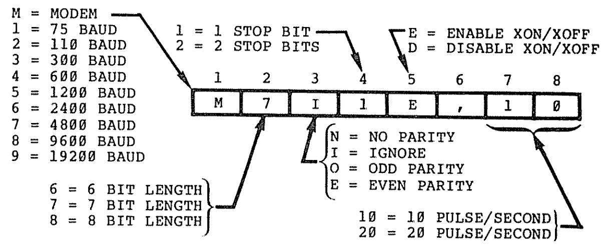
or a similar display. The first line is a shorthand notation for the current communications parameters, which we'll explain shortly.

You can also display the current communications parameters by pressing Function Key F3 (STAT) and ENTER after the Telcom: prompt while in TELCOM. TELCOM will then reply with the current communications parameters.

```
Telcom: Stat
M7I1E, 1Ø pps
```

The communication parameter format is shown in Figure RHSO-1.

Figure RHSO-1 — *Communication Parameter Format*



The first character is the baud rate. This is the speed at which data is transmitted. Use M anytime that you are using the built-in modem for telephone communications (see ACHT-100). Use other rates depending upon the application. Typically, you'd use M for telephone communications (300 baud), and 3 (300 baud) to 9 (19200 baud) for direct computer to computer communication or Model 100 to peripheral device communication.

Character	Meaning
M	Modem (3ØØ baud)
1	75 baud (seldom used)
2	11Ø baud (used on old teletypewriters)
3	3ØØ baud (most common for slow-speed "serial" printers)
4	6ØØ baud (printers, other devices)
5	12ØØ baud (high-speed serial printers, some modems)
6	24ØØ baud (direct computer-to-computer)
7	48ØØ baud (direct computer-to-computer)
8	96ØØ baud (direct computer-to-computer)
9	192ØØ baud (direct computer-to-computer)

The second character is the word length of 6, 7, or 8 bits. Common word lengths for data communications are 7 or 8 bits.

The third character is the parity. Common parity is N for no parity, although I (ignore), O (odd), or E (even) can be used.

The fourth character is the number of stop bits, 1 or 2. One stop bit is most common, although 2 is sometimes used.

The fifth character defines the "XON/XOFF" status. This is a special communications protocol that enables a pause by transmission of a special character. This is normally an E (enabled) for text files, or D (disabled) for other data files.

The next parameter, separated by a comma, is the dialing pulse rate. Some telephone lines support 20 pulses per second when a number is dialed; all support 10 characters per second. Use 20 pps if your local telephone system seems to work all right at this higher rate.

The "default" communications parameters are

```
M7I1E,1Ø pps
```

standing for "modem" (300 baud), 7 bits per word, parity ignored, 1 stop bit, XON/XOFF enabled, and 10 pulses per second. Unless the communications parameters are changed, these parameters will always stay in force.

To set the communications parameters: To set the parameters, enter a line coded in the above format after pressing Function Key 3 (STAT) after a TELCOM prompt:

```
Telcom: Stat 38N1D, 1Ø pps
```

The above example sets the baud rate to 300, number of bits per word to 8, no parity, 1 stop bit, XON/XOFF disabled, and 10 pulses per second.

Check the status by pressing Function Key 3 (STAT), followed by ENTER after the TELCOM prompt.

RLIB-100
Replacing a Line in BASIC, Model 100

The simplest way to replace a BASIC program line: While in BASIC, reenter the line with the same line number and modified commands. BASIC will replace the old BASIC line with the new BASIC line.

Another way to modify a longer line: Use the Edit mode in BASIC. To get into the Edit Mode while in BASIC, enter

```
EDIT
```

This will bring you into the TEXT Editor, and you can edit using the commands in TUHO(100). To get back to BASIC, press Function Key F8. If you want to edit only a portion of the BASIC program enter

```
EDIT line1-line2
EDIT -line2
EDIT line1-
```

These commands establish a "range" of lines for the edit. The first edit is of line1 through line2, the second is of every line from beginning to line2, and the last is from line1 through the end of the program.

Here's one kicker in using the TEXT Editor: The TEXT editor works with ASCII (see ADFW-100) while the BASIC interpreter works with BASIC "tokens" (see TMMO, this section). When going from the BASIC interpreter to the TEXT Editor, the BASIC lines are converted to ASCII; when reentering the BASIC interpreter from the TEXT editor, the ASCII is reconverted. This is all done smoothly, but it does take some time, which might be apparent for longer BASIC programs by the "WAIT" message on the last line of the display.

RNBL-100
Renumbering BASIC Lines, Model 100

BASIC lines can be renumbered on some systems so that additional lines can be inserted, or so that gaps in the line numbering can be deleted to make the listings look "pretty". On all renumbering schemes, all references to the line numbers on GOSUBs and GOTOs will be changed within the program. The renumbered program should run exactly like the old.

The Model 100, unfortunately, has no renumbering capability at this time of writing. Avoid problems by using line increments of 10 or greater — 100, 110, 120, etc.

RNCH
Reading N Characters, BASIC, Model 100

The INPUT$ command lets you read the next n characters from an INPUT. Doing

```
1ØØ A$=INPUT$(1Ø)
```

for example, reads the next 10 characters assigning string A$ to the characters. The characters may be any characters except BREAK. This is a handy command for inputting strings that will contain embedded commas and other "terminators".

The INPUT operation doesn't echo the characters to the screen.

The INPUT$ command when used with a second argument INPUTs from a file (see FMOU, this section).

```
1ØØ A$=INPUT$(1Ø,2)
```

for example, inputs the next 10 characters from the file associated with file number (buffer number) 2. This file number is determined by the OPEN statement for the file (see FMOU, this section).

ROOF-100
Rounding Off in BASIC, Model 100

Read procedure ROOF in the main section. Round-off procedures are identical in the Model 100.

RSWI-100
RS-232-C, What Is It?, Model 100

All of the material in RSWI in the main section applies as well to the Model 100. RS-232-C in the Model 100 uses the standard RS-232-C conventions used by the Model I, II, and III and the same "DB-25" connector. Read RSWI and then read RHSO(100) to find out how to setup and use the Model 100 for telecommunications.

RUTP-100
RUNning a BASIC Program, Model 100

Enter RUN, and your BASIC program will start from the beginning. Enter RUN nnnn, and the program will run from line nnnn.

Enter RUN,R or RUN line#,R and BASIC will leave the currently opened files open. This is handy in running two or more program segments that utilize common files.

R

RWAT
Records, What are They?, Model 100

A file in the Model 100 and other computers is made up of "records". Records are simply arbitrary groupings of data. A classic example is a name, address, and telephone number file. Each of the records within the file consists of one name and its associated address and telephone number. Records within TEXT files such as ADRS.DO (see AHTO, this section) and NOTE.DO (see SCMU, this section) are generally text from the beginning of an entry until an ENTER character — one or perhaps a few lines of information.

SACW-100
Strings, BASIC, Accessing Characters Within, Model 100

Review SHTU in the main section if you're not very familiar with strings.

See procedure SACW in the main section. The information on LEFT$, RIGHT$, and MID$ applies equally well to the Model 100. After you've read that procedure, come back here.

The MID$ command on the Model 100 has an additional function. You can use it as a "replace" function, replacing the characters found by the MID$ with another string that you've defined. To do that simply use MID$ as described, but delete the variable and equals sign on the left and add an equals sign with the replacement string on the right.

Suppose you had a string that you knew had a part number as the 5th through 8th characters, as in:

```
A$="1234Ø122HOW TO DO IT ON THE UNIVAC I"
```

You could easily set the part number to "9999" by:

```
1ØØ MID$(A$,5,4)="9999"
```

Actually, the length parameter is not used. The result always uses the number of characters in the replacement string, so we could have said:

```
1ØØ MID$(A$,5)="9999"
```

The result in this example is:

```
A$="12349999HOW TO DO IT ON THE UNIVAC I"
```

By the way, if you have a Univac I, look for this book in the continuing series of IJG bestsellers. . .

To search a string, use the Model 100 BASIC command INSTR. This command searches for a given smaller string within a larger string. The format is

```
INSTR(start position,larger string,smaller string)
```

To search for "UNIVAC" with A$="12349999HOW TO DO IT ON THE UNIVAC I", do

```
1ØØ B=INSTR(1,A$,"UNIVAC")
```

B will be set to the **character position** of the string, if found. In this case, B would be 29. If the string isn't found, a 0 is returned. Character positions are numbered from 1 to n, left to right.

SBPT-100
Saving BASIC Programs on Cassette, Model 100

BASIC programs can be saved on cassette tape and reloaded at any time.

The two basic commands for doing this are CSAVE and CLOAD.

To save any BASIC program, do a

```
CSAVE "name"
```

This command writes out the current program as a cassette file called "name". File names are from 1 to 6 characters long in the Model 100.

Writing takes anywhere from a few seconds to several minutes, depending upon the length of the program.

You may also do a

```
CSAVE "name",A
```

In this case the file is written out as an "ASCII" file (see AFWA-100). Generally, however, you'll want to do the write without the ASCII option, as the file will be more compact.

After the write, you may do a CLOAD? after rewinding the cassette and repositioning it before the cassette file. The CLOAD? will read in the file, but will not store it into user RAM. It serves only to compare the file on cassette with the program in RAM and verify that it is good. If the comparison is not valid ("Verify Failed" message), you can try rewriting the file. You should have little trouble on the Model 100. (As a matter of fact, you'll probably not use the CLOAD?, but just assume the file is good; as an alternative to CLOAD?, write another copy of the BASIC program on cassette after the first.)

To read in the file, rewind the cassette and position the tape before the file you want to load (use the tape counter or audio output to do this). Now do either a

```
CLOAD
```

or

```
CLOAD "name"
```

to read in either the next file or the specified file called "name".

The file should take the same time to load as it did to write. You should hear the Model 100 speaker emit a high-pitched tone as it starts to read data. When the file is found on cassette, you should see the message "Found:name" where "name" is the file name. If there are one or more files before the one you're loading, you'll see the message "Skip :name", where "name" is the file name of the file being skipped.

If you experience problems in loading cassette files, see CTLC-100.

To load the BASIC file and immediately start execution, add a ",R" after the file name

```
CLOAD "ACCTS",R
```

SCHO
SCHEDL, How to Get to, Model 100

To use SCHEDL, the scheduler, you must first have created a NOTE.DO file by using TEXT (see TUHO, this section). The NOTE file is simply a collection of notes that you have "jotted" down regarding meeting dates, expenses, and so forth — any date related information you care to record. The NOTE.DO file will appear as a file name on the main menu.

If you have created a NOTE.DO file: If you have the main menu on your screen, you'll see the words "BASIC TEXT TELCOM ADDRSS" on the second line and "SCHEDL. . ." on the third line of the screen. Position the shadow area, the cursor, over the word SCHEDL by using the arrow keys to the upper right of the keyboard. Now press ENTER. You should enter SCHEDL and see the display:

```
Schd:
```

If you do not have a NOTE.DO file, then you'll see

```
NOTE.DO not found
```

Press space bar for MENU

See SCMU(100) for information on how to use the SCHEDL program.

SCIB-100

String Concatenation in BASIC, Model 100

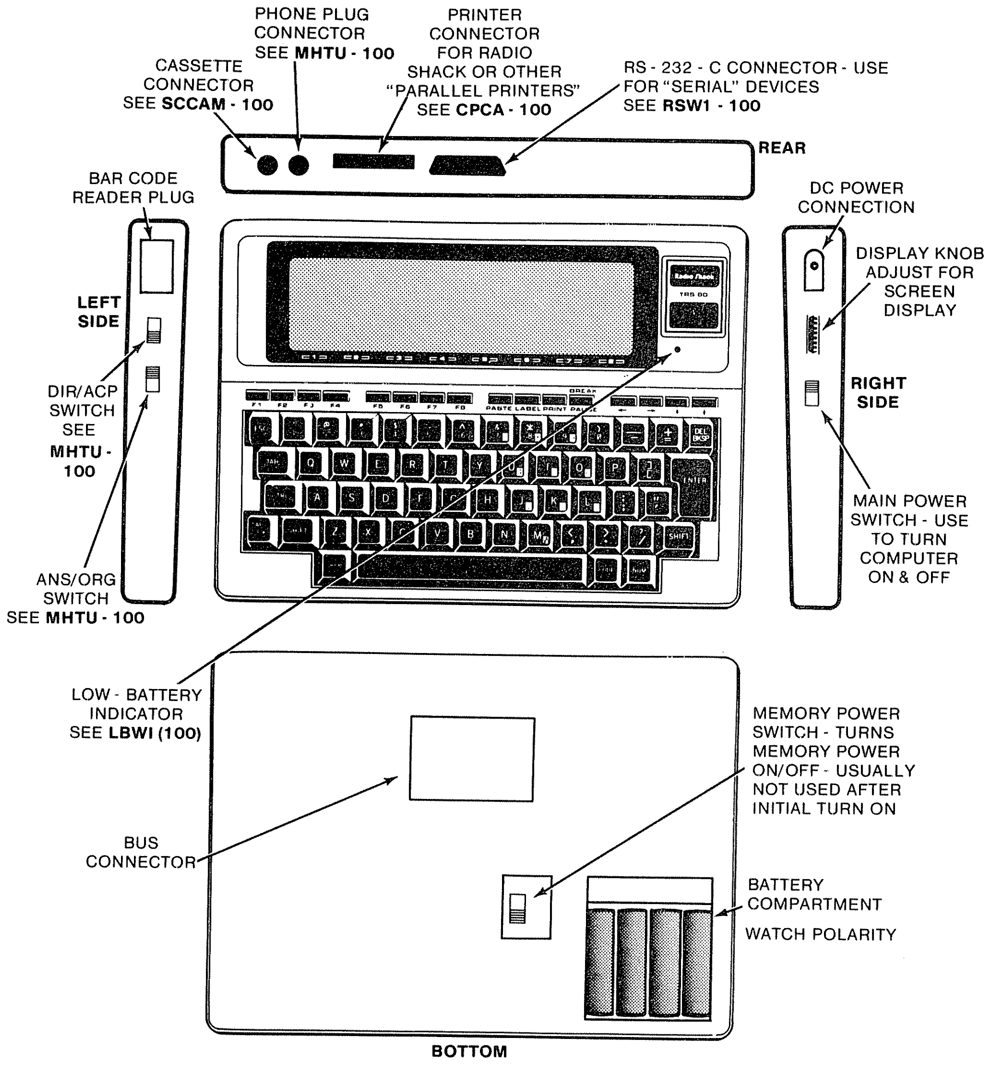
Read procedure SCIB. The information there applies equally well to the Model 100.

SCMO

Switches and Connectors, Model 100

Refer to Figure SCMO-1. Switches and connectors are all over the blasted machine. The "pin-outs" of the connectors are detailed as indicated on the figure.

Figure SCMO-1 — *Switches and Connectors, Model 100*



PHONE PLUG CONNECTOR SEE MHTU - 100
PRINTER CONNECTOR FOR RADIO SHACK OR OTHER "PARALLEL PRINTERS" SEE CPCA - 100
CASSETTE CONNECTOR SEE SCCAM - 100
RS - 232 - C CONNECTOR - USE FOR "SERIAL" DEVICES SEE RSW1 - 100
REAR
BAR CODE READER PLUG
DC POWER CONNECTION
DISPLAY KNOB ADJUST FOR SCREEN DISPLAY
LEFT SIDE
RIGHT SIDE
DIR/ACP SWITCH SEE MHTU - 100
MAIN POWER SWITCH - USE TO TURN COMPUTER ON & OFF
ANS/ORG SWITCH SEE MHTU - 100
LOW - BATTERY INDICATOR SEE LBWI (100)
MEMORY POWER SWITCH - TURNS MEMORY POWER ON/OFF - USUALLY NOT USED AFTER INITIAL TURN ON
BUS CONNECTOR
BATTERY COMPARTMENT
WATCH POLARITY
BOTTOM

SCMU
Schedule Organizer, Using, Model 100

The Schedule Organizer SCHEDL is really nothing more than an applications program that is geared to searching for strings of characters and displaying them on the screen. It operates similarly to Search in TEXT (see TUHO, this section), but displays as many occurrences of the string as it finds, on a line basis.

Suppose that you had the file NOTE.DO, constructed while you were in the executive washroom. It looks like this:

```
12/Ø3/84 #Meeting with Hanley to take over
Amalgamated Disks.
12/Ø3/84 #Can Smith's ---. Not doing effective job.
12/Ø4/84 #Meet with environmentalists about rape of
1Ø5,3ØØ acres of virgin timber.
12/Ø6/84 &Lulu's anniversary - see about fur.
12/Ø7/84 !Check with H. Schmidt at Zurich to find out
new account #.
12/Ø8/84 &Wife's anniversary - candy and flowers.
12/1Ø/84 #Get Lear Jet refueled.
```

The file consists of a number of entries. Each entry is terminated by an ENTER and consists of one or more lines. Another word for the entry is a "record".

You could easily use SCHEDL to find and list all personal notes, such as "&Lulu's anniversery — see about fur.", all business notes, such as "#Meeting with Hanley. . .", or all urgent notes, such as "!Check with H. Schmidt at Zurich. . .". You could also have it list all dates of 12/03/84.

There's no mystique about SCHEDL, then — it's simply a way to go through a large file of notes called NOTE.DO and pull out the ones that you're interested in, grouped by special symbol or date. It's akin to searching through a stack of 3 inch by 5 inch index cards on which you've written appointments, notes, names, etc. The SCHEDL program is the device by which all of the appropriate cards are pulled out and laid down in a row for you to see.

See SCHO(100) to find out how to get into SCHEDL.

To find information in an existing NOTE.DO file:

1. After the

```
Schd:
```

prompt, press F1.

2. You'll now see

```
Schd: Find
```

3. Enter a string of characters to be found. If you wanted a list of all 12/03/84 items, for example, you might enter 12/03/84 or 12/03. In this example, you'd then see:

```
Schd: Find 12/Ø3
12/Ø3/84 #Meeting with Hanley to take
over Amalgamated Disks.
12/Ø3/84 #Can Smith's ---. Not doing
effective job.
```

The "case" — upper or lower — is ignored in this search.

4. At this point either all lines with the "Find" string have been displayed, or there are too many lines to display on the screen. In the latter case, you'll see the message:

```
More Quit
```

above Function Key indicators F3 and F4. To see more of the list press Function Key F3. To quit, press Function Key F4.

5. When all items have been displayed, the prompt message

```
Schd:
```

will reappear.

As a second example of searching, suppose that you had wanted to find all personal notes, indicated by the special character "&". Entering

```
Schd: Find &
```

would have resulted in

```
12/Ø6/84 &Lulu's anniversary - see about fur.
12/Ø8/84 &Wife's anniversary - candy and flowers.
```

To scan the entire NOTE.DO file:

1. After the

```
Schd:
```

prompt, press F1.

2. You'll now see

```
Schd: Find
```

3. Press ENTER. This essentially says, find *everything*.

4. The first lines of the NOTE.DO file will then be displayed. At this point either all lines of the NOTE.DO file have been displayed, in a short file, or there are too many lines to display on the screen. In the latter case, you'll see the message:

```
More Quit
```

above Function Key indicators F3 and F4. To see more of the list press Function Key F3. To quit, press Function Key F4.

5. When all items have been displayed, the prompt message

```
Schd:
```

is again displayed.

To list items on the printer: Do the same thing as in the two procedures above, but press Function Key F5 (LFND) instead of Function Key F1 (FIND). The list will be printed on the system line printer without pauses.

To update the NOTE.DO file: Use the TEXT program operating on File NOTE.DO. See TUHO(100).

SFLO-100
Strings, Finding Length of, BASIC, Model 100

Read SFLO in the main section. Information on string lengths applies to the Model 100 as well.

SGBA
Spaces, Generating in BASIC, Model 100

The SPACE$(nn) command works like a special case of STRING$(nn,char); it generates nn space (blank) characters.

```
100 PRINT A$;SPACE$(20);B$
```

for example, prints string A$, 20 spaces, and string B$.

SHTG
Sound, How to Generate, Model 100

Using the BASIC command BEEP causes a high-pitched beep for about 1/2 second. This is the same beep used by the 100 to alert the user to error conditions.

Use the

```
SOUND ON
```

or

```
SOUND OFF
```

command to enable or disable a beep when loading from cassette or waiting for a carrier signal when using the modem.

Using the BASIC SOUND command lets you generate a wide range of sounds with different durations. The format of SOUND is

```
SOUND pitch,length
```

where pitch is 0 to 16383 and length is 0 to 255.

Each count in the length parameter is about 20/1000ths of a second (50 counts is about one second).

Larger counts in pitch produce higher pitches; smaller counts produce lower pitches. Try pitch counts from 100 (inaudible except to my dog, Cursor) through 16383 (very low).

Both the pitch and length counts are "linear", that is, double the count and you'll have twice the length or one-half the pitch. Musical octaves double in pitch by the way, so it's not too hard to construct musical notes using SOUND.

SHTU-100
Strings, BASIC, How to Use, Model 100

All of the information in SHTU applies equally well to the Model 100. Use the DEFSTR as in the Models I/II/III to define name ranges for string variables.

SIPS-100
Semicolon, In BASIC PRINTs, Model 100

Read the information about semicolon use in procedure SIPS. It applies equally well to the Model 100.

SLTF
Saving and Loading TEXT Files, Model 100

Follow the steps in TUHO, the procedure on using TEXT. Even though you may initially position the cursor over the file to be SAVEd, you'll be entering TEXT. Ditto LOADing TEXT files.

SMLO
Saving a Machine Language Program, Model 100

You can save a machine language program to RAM or CASsette from memory. Of course, the machine language program is in RAM already, but saving it to RAM saves it as a machine language "file". The file may also simply be data and not machine language instructions.

The machine language program may have been assembled by using an external 8085 assembler (there's none for the Model 100 currently), POKEd into RAM at a specific location, and then debugged. We can't tell you how to do that here, and would suggest looking at the many 8080/8085 assembly language programming books available.

To save a machine language program in memory to RAM, do a

```
SAVEM"RAM:filename",start,end,entry
```

or simply

```
SAVEM"filename",start,end,entry
```

where filename is a standard Model 100 filename (see FNMO, this section), start is the start address of the program in decimal, end is the end address of the program in decimal, and entry is an optional execution address of the program in decimal.

A good extension to use on the filename is .CO, for "command file", although the extension is optional.

To save to cassette, use the same form with the CAS device designator:

```
SAVEM"CAS:filename",start,end,entry
```

Better yet, use the CSAVEM command, logically equal to the SAVEM"CAS:filename" command:

```
CSAVEM"filename",start,end,entry
```

The two commands

```
SAVEM"RAM:SCREEN.CO",52000,52200,52000
CSAVEM"SCREEN",52000,52200,52000
```

save the area of memory from 52000 through 52200 to a RAM file called SCREEN.CO and to a cassette file called SCREEN. The execution address is 52000.

SNER-100
SN Error, Model 100

See SNER in main section.

SPHU-100
Single-Precision Variables in BASIC, How to Use, Model 100

Variables in the Model 100 are double-precision variables (8 bytes) unless defined as single-precision by a "%" suffix, as in A%. Single-precision variables take up less room in memory but have only 6 decimal digits of precision compared to 14 for double-precision variables.

A range of single-precision variables may be specified by the DEFSNG command. DEFSNG A-G, for example, specifies all variables starting with A through G as single precision; AS, FD, and GG would be single-precision in this case.

To define a variable as single-precision, use the suffix "!". For example, in the following code, A! is a single-precision variable:

```
110 A!=123.456
```

Using a suffix of "E" also denotes a single-precision number, in this case with scientific notation (see AOIB-100).

```
100 A=123.45E+17
```

SSFA-100
String Space, Finding Amount Left, Model 100

FRE(A$) returns the amount of free string space. The expression inside the parentheses may be any string constant ("ONE TANDY") or variable, even an undefined variable. Doing a 100 PRINT FRE(""), for example, will print the free string space remaining. The string space area is established by a CLEAR statement (see OSSH). As strings are manipulated, this string space is used up; FRE allows you to check string space availability either within a program or in the command mode to see how much space is left.

SSFO
Scratch, Starting From, Model 100

Pressing CTRL and BREAK/PAUSE together while turning on the power switch will clear *everything* on the Model 100 — all the RAM files, TIME$, DAY$, DATE$, etc. It's similar to switching the MEMORY POWER switch OFF and then ON again. Do this only if you've "backed up" your critical programs. This is known as a "cold start".

Normally, the Model 100 will start from a "warm start". All previous files, date, day, and time will still be intact, thanks to the Ni-Cad battery backup.

Good thing that you have to be an extra-terrestial to reach those "cold-start" keys together, eh? (Better than, say, SHIFT, A.)

STAC-100
Stack, Description, Model 100

Read procedure STAC in the main section to see how the stack area is used in both BASIC or machine language. The equivalent PUSH, PULL, CALL, and RET instructions are also found in the Model 100.

STER-100
ST Error, Model 100

See STER in main section.

STRD-100
STRING$ Command, BASIC, Model 100

Read STRD in the main section. STRING$ works identically for the Model 100 as in other systems.

SUBB-100
Subroutines, BASIC, Model 100

Read procedure SUBB in the main section; information on subroutines applies equally well to the Model 100.

SUBP-100
Speeding Up Your BASIC Programs, Model 100

Read procedure SUBP in the main section. All of the points brought up there apply as well to the Model 100. After you read SUBP, come back here.

Point 8: graphics are handled differently on the Model 100. See GHSO(100) for an explanation of Model 100 graphics.

In addition (point 9), as of this time there is no assembler for the Model 100. The Model 100 uses a CMOS version of the 8085 microprocessor, which, in turn, is an improved 8080 microprocessor. Assembly language is the ultimate speed up for any application, and I'm certain there will be an assembler for the Model 100, either provided by Radio Shack or others.

TCML-100
Transferring Control to a Machine Language Program from BASIC, Model 100

See procedure LMRO(100). The best way to transfer control to a machine language program from BASIC is to load the machine language program using the RUNM command, which loads a RAM file and then executes it, starting from the execution address specified in the file (you do not have to know what it is).

The BASIC CALL command can also be used to transfer control to a machine language **subroutine** in memory. This normally would not be used for a complete machine language program which runs independently from BASIC, however. See procedure RCWA-100.

TCNU-100
Two's Complement Numbers, Using, Model 100

Two's complement numbers are also used in the Model 100 in exactly the same fashion as in other Radio Sh:.ck computers. The microprocessor here, of course, is the 80C85, but both the assembly language and BASIC use of two's complement numbers are identical to use on the other systems. Read procedure TCNU in the main section.

TEHT
TELCOM, How to Get to, Model 100

If you have the main menu on your screen, you'll see the words "BASIC TEXT TELCOM ADDRSS" on the second line of the screen. Position the shadow area, the cursor, over the word TELCOM by using the arrow keys to the upper right of the keyboard. Now press ENTER. You should now be in TELCOM and see the display:

```
M7I1E,1Ø pps
Telcom:
```

Read procedure ACHT-100 or MHTU-100 to find out how to use TELCOM.

THTO
TEXT, How to Get to, Model 100

If you have the main menu on your screen, you'll see the words "BASIC TEXT TELCOM ADDRSS" on the second line of the screen. Position the shadow area, the cursor, over the word TEXT by using the arrow keys to the upper right of the keyboard. Now press ENTER. You should now be in TEXT and see the display:

File to Edit?

Now read procedure TUHO(100) to find out how to use TEXT.

TIBP-100
Time, in BASIC Program, Model 100

The TIME$ function gets the time in the format HH:MM:SS.

Of course, the current time must be valid; this means that the time must have been entered by doing a TIME$="HH:MM:SS" while in command or program mode. Note that a 24-hour clock is used — 13:00 is 1:00 pm, 14:00 is 2:00 pm, etc. Note that even when the power switch is off, the Model 100 updates both time and date.

TIMO
Time Interrupt, BASIC, Model 100

The Model 100 contains a "real-time-clock" that counts the actual time and changes calendar days just like your quartz wrist watch. See TIBP-100 and DIBO(100) to set the time, date, and day of the week. You can create a "real-time-clock" interrupt in your BASIC program so that at a certain time a BASIC subroutine will be entered. This is a neat feature of the Model 100, even though you probably won't use it often. To GOSUB line 10000 at noon, for example, you'd have this line at the beginning of your program:

```
1ØØ ON TIME$="12:ØØ:ØØ" GOSUB 1ØØØØ
```

If you then did a TIME$ ON command, at precisely noon BASIC would enter the subroutine at line 10000. Because a GOSUB is executed, you can always get back to the "interrupted point" by doing a RETURN at the end of the 10000 subroutine.

What's in the 10000 subroutine? Whatever you want — maybe you want to sound a midday beep or even automatically dial a Bulletin Board!

To disable the ON TIME$ interrupt, do

```
TIME$ OFF
```

To enable it again, do

```
TIME$ ON
```

To disable the ON TIME$ interrupt, but cause the Model 100 to remember the interrupt occurred, do

```
TIME$ STOP
```

When the interrupt is again enabled by TIME$ ON, the TIME$ STOP command causes an immediate GOSUB action to the ON TIME$ subroutine. Get it? Suppose that you had a noon time interrupt, but disabled it by TIME$ OFF at 11:55. If you enabled it by TIME$ ON at 12:05, nothing would happen. However, if you had disabled it by TIME$ STOP at 11:55 and again enabled it by TIME$ ON at 12:05, an immediate GOSUB 10000 would occur after the TIME$ ON.

A weird set of commands you say? I would have given my eyeteeth for such a neat "real-time-clock" interrupt on many minicomputer systems 10 years ago. Here's an "armed" and "enabled" **actual time** interrupt in a portable computer! Fantastic.

TMER-100
TM Error, Model 100

See TMER in main section.

TMMO
Tokens in BASIC, Model 100

Model 100 BASIC uses a special form of encoding for BASIC commands. BASIC commands such as LIST, NEW, FOR, RETURN, and all others are not represented by their ASCII (see ADFW-100) equivalent text, but as one-byte codes of 128 through 255. This decreases the amount of memory storage required for BASIC programs. Tokens are always used in in-memory storage of BASIC programs, but BASIC programs may optionally be stored on cassette as ASCII files, and BASIC files written to the communications line are always ASCII, with "straight text" and no tokens.

The token codes used in Model 100 BASIC are shown in Table TMMO-1 below.

Table TMMO-1 — *Model 100 Tokens*

TOKEN VALUE	TOKEN	TOKEN VALUE	TOKEN	TOKEN VALUE	TOKEN
128	END	171	DATE$	214	OR
129	FOR	172	DAY$	215	XOR
130	NEXT	173	COM	216	EQV
131	DATA	174	MDM	217	IMP
132	INPUT	175	KEY	218	MOD
133	DIM	176	CLS	219	\
134	READ	177	BEEP	220	>
135	LET	178	SOUND	221	=
136	GOTO	179	LCOPY	222	<
137	RUN	180	PSET	223	SGN
138	IF	181	PRESET	224	INT
139	RESTORE	182	MOTOR	225	ABS
140	GOSUB	183	MAX	226	FRE
141	RETURN	184	POWER	227	INP
142	REM	185	CALL	228	LPOS
143	STOP	186	MENU	229	POS
144	WIDTH	187	IPL	230	SQR
145	ELSE	188	NAME	231	RND
146	LINE	189	KILL	232	LOG
147	EDIT	190	SCREEN	233	EXP
148	ERROR	191	NEW	234	COS
149	RESUME	192	TAB(	235	SIN
150	OUT	193	TO	236	TAN
151	ON	194	USING	237	ATN
152	DSKO$	195	VARPTR	238	PEEK
153	OPEN	196	ERL	239	EOF
154	CLOSE	197	ERR	240	LOC
155	LOAD	198	STRING$	241	LOF
156	MERGE	199	INSTR	242	CINT
157	FILES	200	DSKI$	243	CSNG
158	SAVE	201	INKEY$	244	CDBL
159	LFILES	202	CSRLIN	245	FIX
160	LPRINT	203	OFF	246	LEN
161	DEF	204	HIMEM	247	STR$
162	POKE	205	THEN	248	VAL
163	PRINT	206	NOT	249	ASC
164	CONT	207	STEP	250	CHR$
165	LIST	208	+	251	SPACE$
166	LLIST	209	-	252	LEFT$
167	CLEAR	210	*	253	RIGHT$
168	CLOAD	211	/	254	MID$
169	CSAVE	212	^	255	'
170	TIME$	213	AND		

TOCH-100
Turning on the Computer, How To, Model 100

Out of the box:

1. Insert 4 AA alkaline batteries into the battery compartment (see Figure SCMO-1). I'd suggest using expensive batteries; in general, they are more cost effective than less expensive types.

2. Turn on the MEMORY POWER switch on the underside of the unit. This supplies power to the memory of the unit. Leave this switch on unless you want to erase everything stored in memory. It can be left on even when installing new batteries.

3. Turn the ON/OFF switch on the right-hand side of the unit to ON.

4. You should now see the main "menu" on the screen.

Don't see it? There's a knob marked DISP on the right side of the unit. This controls the contrast of the lcd (liquid crystal display). Turn it counterclockwise to increase the contrast, or, as the RS manual says "optimum viewability" (UGH!). Go to BHTO(100) to continue.

After initial power up: Simply turn ON the ON/OFF switch on the right-hand side of the unit and adjust the DISP knob on the side for optimum viewability (see UGH above). Don't worry about leaving the unit on; it will shut itself off after a short time (see POOM, this section) lest you forget.

After a power down: See POOM(100). Turn ON/OFF switch on right side OFF, then ON.

If you have optional equipment attached, it's a good policy to turn on that equipment after the Model 100 is turned on, and off before the Model 100 is turned off.

TSPD-100
To Stop the Display in BASIC, All Systems, Model 100

If your BASIC program is tracing or listing rapidly and things are going by too fast, stop the display by pressing the BREAK/PAUSE key (without the SHIFT). This PAUSEs BASIC program execution. Pressing BREAK/PAUSE again restarts execution.

TSTP-100
To Stop the Program, Model 100

Press SHIFT followed by BREAK/PAUSE. The BASIC program will stop. Can't get the computer to recognize any other key? You pressed BREAK without the SHIFT, which is really a PAUSE; no key will be recognized. Press BREAK/PAUSE again without a SHIFT. The program should resume. Now press SHIFT, BREAK and you'll see a

```
Break in 11Ø
```

or similar statement, indicating that the BASIC program was interrupted.

Also read procedure TSTP in the main section.

TUHO
TEXT, Using, Model 100

See THTO (100) to find out how to get into the TEXT program.

The TEXT program is an application program permanently resident in ROM. It allows you to create and modify "text" or ASCII files. For background on ASCII files, see AFWA-100.

The TEXT program contains about 50% of the capability of a typical word-processing program. It can be used fairly effectively as a word processor, although it won't have all the "bells and whistles". Most files used by other 100 programs, such as SCHEDL (see SCMU, this section) and ADDRSS (see ADMO, this section) are first created using TEXT.

To create a new file:

1. Get into TEXT by THTO(100).

2. You'll now see the message

```
File to Edit?
```

displayed on the screen. Enter a 1 to 6 character name for the file. The file name extension ".DO", for document, will be added to the file name (see FNMO, this section).

3. The prompt message will disappear, and you'll see a blinking cursor in the upper left-hand corner of the screen.

4. Typing text now will generally create a new file. At the end of a line, words will "wrap around" to the next line if they are too long to fit within the 40-character limitation.

5. Exit TEXT by pressing Function Key F8. It will take you back to the main menu. When you do so, note that the file you created will have its name displayed on the menu.

6. To perform other editing actions, see the remainder of this procedure.

To edit an existing file: You can edit an existing file by using TEXT in one of two ways:

1. Get into TEXT by THTO(100).

2. You'll now see the message

```
File to Edit?
```

displayed on the screen. Enter the 1 to 6 character name for the file. The file name extension ".DO", for document, will be added to the file name (see FNMO, this section), so you won't have to enter this part of the name.

3. The file will now be loaded into TEXT, and you can continue editing with the remainder of these procedures.

The second way to get into TEXT is to select the file from the main menu. Move the cursor (the shaded area) over the file name you wish to edit with the arrow keys in the upper right-hand portion of the keyboard. Now press ENTER; the file you've selected will be loaded into TEXT, and TEXT will start execution. See the remainder of this procedure for editing actions.

To see the Function Key actions: Press the LABEL key (upper row). The actions for each of the 8 Function Keys will be displayed on the last line of the screen. You can keep this "legend" if you wish, or get back to a full-screen display by pressing LABEL again.

To enter text into TEXT: Jus' keep typing. No need to worry about where the words fall, TEXT will "wrap-around" the words from line to line (it won't "split up" words on two lines).

To see the beginning, end, or any portion of the file: Use the Up Arrow and Down Arrow keys (upper right of keyboard) to "scroll" the text up or down. You can do this at any time. Holding down the key will cause continuous scrolling. To get to the very beginning of the text, press the CTRL key, followed by Up Arrow. To get to the very end of the file, press the CTRL key, followed by Down Arrow.

Entering SHIFT, Up Arrow or Shift Down Arrow will display a "screen full" of text at a time, moving up or down.

To move the cursor along the line:

Use the Left and Right Arrow keys to position the cursor along the line. Note that holding down the key will cause a continuous move along the line. Note also that the cursor position will move from line to line. Pressing CTRL followed by Left or Right Arrow will move to the beginning or end of the current line.

To move the cursor a word at a time:

Press the SHIFT key followed by Left or Right Arrow. The cursor will move to the beginning of the previous or following word.

To insert text at any point in the file:

1. Position the cursor by the Arrow keys as described above. The cursor should be on the character before which you want text inserted. Example: If you have the line

```
Now is thè time. . .
```

and want to change it to

```
Now is not the time. . .
```

move the cursor on top of the "t"in "the".

2. Now type in the new text. In this case, you'd type in "notb", where b is a blank.

3. To get out of this insert mode, move the cursor with the Arrow keys.

4. You're really in this insert mode at all times. To "type over" existing text, you must do a "delete" first, and then an "insert".

5. To insert text at the end of the file, do a CTRL followed by Down Arrow, or move the cursor manually to the end and enter text.

To delete a few characters from the file:

1. Position the cursor to the deletion point as described above. The deletion point is the first character to be deleted.

2. Press SHIFT, followed by DEL/BKSP (upper right corner).

3. The cursor will remain stationary, the character will be deleted, and the remaining text will "snake up", ready for deletion of the next character.

4. Repeat as often as necessary.

Another way to delete: Use the BKSP key (DEL/BKSP unshifted). In this case, the cursor must be immediately to the right of the character you want to delete.

To delete larger blocks of text:

1. Position the cursor to the first character to be deleted by the above procedures.

2. Press Function Key F7 (SELECT).

3. Now move the cursor anywhere. As it moves, you'll see a shaded area (reverse video) defining a "block" of text. Make this block as large or small as you want by moving the cursor.

4. Now press Function Key F6 (CUT). The text will be deleted, and the remainder of the text will "snake up".

5. Note that this text has been deleted from the deletion point, but that it still remains in the "Paste" buffer. It can be "Copied" until a new block of text overwrites it.

6. In positioning the cursor to define the block in step 3 above, you can use various keys for positioning the cursor described above — the Arrow keys, SHIFT, Left or Right Arrow for words, etc.

To move a block of text:

1. Follow steps 1 through 4 above to delete the text from a point in the file.

2. At this point you now have the text in the "Paste" buffer. This buffer will hold one character or thousands of characters, but there is one catch — there is only *one* Paste buffer. Doing any other "Cut" operation will overwrite and destroy the previous Paste buffer text.

3. Move the cursor by the procedures above to the point in the text you want the block inserted. The text in the Paste buffer will be inserted *before* this point.

4. Press the PASTE Key (upper row). The text from the Paste buffer will be inserted at the cursor point in the text.

To repeat a block of text many times throughout a file:

1. Follow the steps above for the first move.

2. Position the cursor to the second point in the text.

3. Press PASTE.

4. Repeat for the other points at which you want the text copied.

To copy a block of text:

1. Position the cursor to the first character to be deleted by the above procedures.

2. Press Function Key F7 (SELECT).

3. Now move the cursor anywhere. As it moves, you'll see a shaded area (reverse video) defining a "block" of text. Make this block as large or small as you want by moving the cursor.

4. Now press the Function Key F5 (COPY). The text will not be affected, but will be moved to the "Paste" buffer. It can be "Copied" until a new block of text overwrites it.

5. In positioning the cursor to define the block in step 3 above, you can use various keys for positioning the cursor described above — the Arrow keys, SHIFT, Left or Right Arrow for words, etc.

6. Move the cursor by the procedures above to the point in the text you want the block inserted. The text in the Paste buffer will be inserted **before** this point.

7. Press the PASTE Key (upper row). The text from the Paste buffer will be inserted at the cursor point in the text.

To copy a block of text many times throughout a file:

1. Follow the steps above for the first copy.

2. Position the cursor to the second point in the text.

3. Press PASTE.

4. Repeat for the other points at which you want the text copied.

To find a string of characters: Suppose that you have a long TEXT file and want to find all occurrences of the string "SISAL". Do this:

1. Move the cursor by the above procedures to the start of the point at which you wish the search to begin. If you want to search the entire file, move the cursor to the beginning of the text by CTRL, followed by Up Arrow.

2. Press Function Key F1 (FIND).

3. You'll now see the message

```
String:
```

displayed on the last active line of the screen. Enter the string you want to search for. The string may be 1 to any number of characters long.

4. If the string is not found, the message

```
No match
```

will be displayed.

5. If the string is found, the file will be scrolled (if necessary) and the cursor will be positioned over the found string.

6. To find the next occurrence of the string, repeat steps 2 through 5 again. You won't have to retype the string, as it will be displayed after the "String" prompt.

7. Note that the search ignores whether the string is in upper or lower case. In other words, if the search string is "SISAL", the text "Sisal", "SISAL", and "sisal" will be found.

Using the Search function to define a block: You can use the Search Function to define a block for a move or copy. After a SELECT (Function Key 7), do a Search Function to find the string. If the string is not found, the cursor remains where it is. If the string is found, the block (shaded or reverse video area) becomes all characters from the initial cursor position up to but not including the string.

To get back to the main menu: Press Function Key F8 (or press the ESC key twice). This brings you back to the main menu. Important: Any modification you did to the file while in TEXT is now a permanent part of the file!

Closing a file: See previous paragraph. You've "closed" the RAM file by getting back to the main menu. "Closing" a file simply means terminating the file properly and picking up the odds and ends. In this case, it means simply getting back to the main menu, as the file is modified "on the fly" as you're using TEXT.

To print a file:

1. Connect the Model 100 to a parallel printer via the back connector.

2. "Ready" the printer. Power should be on, paper properly put in place, and the printer should be "on line", usually indicated by a light somewhere on the printer.

3. To print out the entire file, press SHIFT, followed by PRINT (upper row).

4. The prompt message

```
Width:
```

will now be displayed. If you have ever entered the width of the printer in columns, that number will also be displayed. To keep the same width, press ENTER, otherwise, enter the number of characters per line for your printer, followed by ENTER.

5. The entire file should now be printed.

6. To print out only what appears on the display, press PRINT. (The normal function, of course, would be SHIFT, PRINT.) Note that this will print even non-file text, such as the Function Key labels.

The Print function will not "right justify" the lines — you'll have a "ragged right" edge, but it will "word wrap" the text so that words are not split up among two lines.

To save a file to cassette:

1. Connect a cassette recorder (see CRPI-100).

2. Rewind the cassette tape to the spot at which you want to place the file. If this is a previously used cassette tape, you might want to erase a portion of the tape directly before the write point by doing a RECORD, PLAY manually (temporarily pull out the "REM" plug to control tape movement; use the tape counter to record the position).

3. Press Function Key F3 (SAVE).

4. You should now see the prompt message

```
Save to:
```

displayed on the screen.

5. Enter a 1- to 6-character name for the name of the cassette file you'd like the text stored under.

6. Up to this point, there should have been no tape motion. When the name is entered, you should hear the

click of the internal Model 100 relay as the Model 100 TEXT program turns the recorder on and off. At the same time, you should see the tape move as the file is written out.

7. When the file has been written, the prompt message will disppear.

To load a file to cassette:

1. Connect a cassette recorder (see CRPI-100).

2. Rewind the cassette tape to the spot at which the file should be (use the tape counter). If you do not know where the file is on the tape, but know that it's somewhere on the tape, rewind the tape to the beginning.

3. Before you do anything else: Do you want the file that is to be loaded "appended to" the text already in memory? If so, continue. If not delete all of the current text by doing a block delete (see above), or simply get back to TEXT by pressing F8 to get back to the main menu, and then returning to TEXT.

4. Press Functi၁n Key F2 (LOAD).

5. You should now see the prompt message

```
Load from:
```

displayed on the screen.

6. Enter a 1- to 6-character name for the name of the cassette file you want to load.

7. Up to this point, there should have been no tape motion. When the name is entered, you should see the tape move as the data is read in. At the same time you should hear a high-pitched sound as the tape is read. You'll also see any file name that comes before the file you're looking for displayed as

```
Skip: XXX
```

where XXX is the skipped file name. When the file you've specified is found, you'll see

```
Found:XXX
```

displayed on the screen, and the file will be loaded.

Want to quiet that high-pitched sound as the tape is read? See special coupon at the back of this manual for IJG/Remington-Rand TRS-80 Model 30-06. Or use SOUND OFF.

8. If you have loaded with no text in memory, you now have a new file in memory with the name you specified when you originally entered TEXT. If you appended a file by not deleting the text before the Load, you now have a larger file under the TEXT name.

To cancel any operation: Press SHIFT, followed by BREAK/PAUSE. You'll either see the normal TEXT display with the cursor, or a message "Operation Aborted".

To tab: Use CTRL, followed by I. Tabs are character positions 0, 8, 16, 24, and 32 (counting from 0).

Other CTRL sequences: There are other CTRL sequences that duplicate functions defined above. See the Model 100 manual for these. You may choose not to use these. If God had wanted us to use CTRL, U for editing, he wouldn't have given us Function Key 6 (CUT).

UEER-100
UE Error, Model 100

See UEER in main section.

ULER-100
UL Error, Model 100

See ULER in main section.

UTFB-100
Using Trigonometric Functions in BASIC, Model 100

Read procedure UTFB in the main section. It applies exactly to the Model 100.

notes

VLFI
Variable Location in BASIC, Finding. Model 100

To find the location of a variable in BASIC, use the VARPTR function. It returns the location of the actual start of numeric data for numeric variables. If you had

```
100 A=1234
110 PRINT VARPTR(A)
```

for example, VARPTR(A) would return the location of the data for variable A:

```
 90 I=0
100 A=1234
110 FOR I=VARPTR(A) TO VARPTR(A)+7
120 PRINT PEEK(I)
130 NEXT I
```

This data is in "bcd" or binary-coded-decimal format, and may not be easily recognizable for single- or double-precision numbers. Integer variable data consists of two bytes in "8085" format (see EFAF, this section).

When VARPTR is used with string variables it returns the address of a string parameter block, consisting of three bytes. The first is the length of the string. The second and third are the address of the string in "8085" format.

```
 90 I=0:B=0
100 A$="XYXX"
110 B=PEEK(VARPTR(A$)+2)*256+PEEK(VARPTR(A$)+1)
120 FOR I=0 TO PEEK(VARPTR(A$))-1
130 PRINT CHR$(PEEK(B+I))
140 NEXT I
```

One word of caution. The address returned by VARPTR will be negative if the variable is located above 32768, and all variables are (see MMMO, this section). This value will work in PEEKs, but to see the actual address, use 65536+VVVVV, where VVVVV is the value returned by the VARPTR.

Another word of caution. If you use VARPTR, use it **directly** before the variable you're after, and don't introduce any new variables between the VARPTR and the use of the address returned. As new variables are added to a BASIC program, or as the BASIC program is modified, the locations of variables **move**!

SOFTWARE

IT'S JUST GREAT

ON SALE NOW

IJG

EXTRA! EXTRA!

50,000 Word Dictionary and Spelling Correction for TRS-80 Model I & III. Electric Pencil 2.0z' Integrated Word Processing System!

Dateline: January 1983.

IJG has done it again.

First, IJG brought the world the Electric Pencil 2.0z Word Processing System. The easy to learn, easy to use word processing system considered by many to be one of the finest available.

Now, IJG brings to the Electric Pencil system . . .

Blue Pencil, the Expandable 50,000 Word Dictionary.

That's right. A 50,000 word expandable dictionary that becomes an integral part of your Electric Pencil.

A virtually unlimited number of new words can be added to your Blue Pencil, making Blue Pencil especially useful for proper names, acronyms and professional terms.

No need to exit the operating system or run any other programs, just a simple two key command and Blue Pencil is quickly checking your text.

It's so simple.

Spelling and corrections are then easily accomplished by using Electric Pencil's search-and-replace feature. Or, if you would like your system to do all the work, there also is . . .

Red Pencil, Easy Spelling Correction.

Red Pencil brings together all the features of Electric Pencil and Blue Pencil, with a flexible spelling correction program that makes copy proofing a breeze.

Instead of dropping you off to edit and correct your own text, your enhanced Pencil will now go down the list of misspelled words with you, word by word, and offer you several options from a menu.

You can type in the correct spelling, leave the word alone, display the word in the sentence and then make your correction, display the dictionary at the nearest location and scroll up or down to find the correct spelling, or add or delete the word in the dictionary.

Then Red Pencil returns you to your Electric Pencil file with all your corrections present in the text, and your new file saved under a special name on the data disk. Your original file is untouched.

Fast and Easy Proofing.

Both Red Pencil and Blue Pencil are available (disk only) for $89.95 each at computer stores, B. Dalton Booksellers and independent book dealers around the world.

Along with the Electric Pencil 2.0z Word Processing System for $89.95 (disk version).

If IJG products are not available from your local dealer, order direct from IJG.

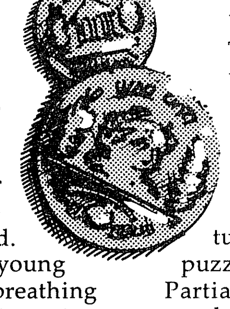
Include $4.00 for shipping and handling. Foreign residents add $11.00 plus purchase price. U.S. funds only please.

IJG, Inc. 1953 West 11th Street
Upland, California 91786
Phone: 714/946-5805

IJG

Helping You Help Yourself.

WANTED: Dragon Slayer. No experience necessary, will train.

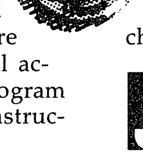


Explore Adventure. Down through the ages the exploits of brave adventurers on missions of high intrigue and danger have captured the imaginations of young and old. Whether it be a fearless young warrior out to bring a fire breathing dragon his due, or a scientist racing time to save a galaxy, they have become the fantasies and daydreams of us all, but few of us have actually dared to embrace these fantasies. Here, perhaps, is your chance . . .

The CAPTAIN 80 book of BASIC Adventures.

Adventure is an intelligent, narrative program which presents an unsolved puzzle for the would-be hero to attempt to solve. The program recognizes words and communicates with the adventurer in the form of descriptive phrases, identifying the circumstances and giving clues to which the adventurer then reacts. Solving the puzzle leads to rewards by degree. Partial solutions earn partial rewards. Find all the treasure and you win the game and all the glory that goes with it.

BASIC Adventure program generator.



Included also in this 252 page book is the Adventure Generator. This program will actually write another BASIC program around your imaginative instructions (requires disk BASIC).

Although written for the TRS-80 Models I & III, these challenging adventure programs will easily adapt to most Microsoft BASIC compatable systems.

If it's from IJG, It's Just Great!

The CAPTAIN 80 BOOK OF BASIC ADVENTURES is available for $19.95 at computer stores and independent book dealers around the world.

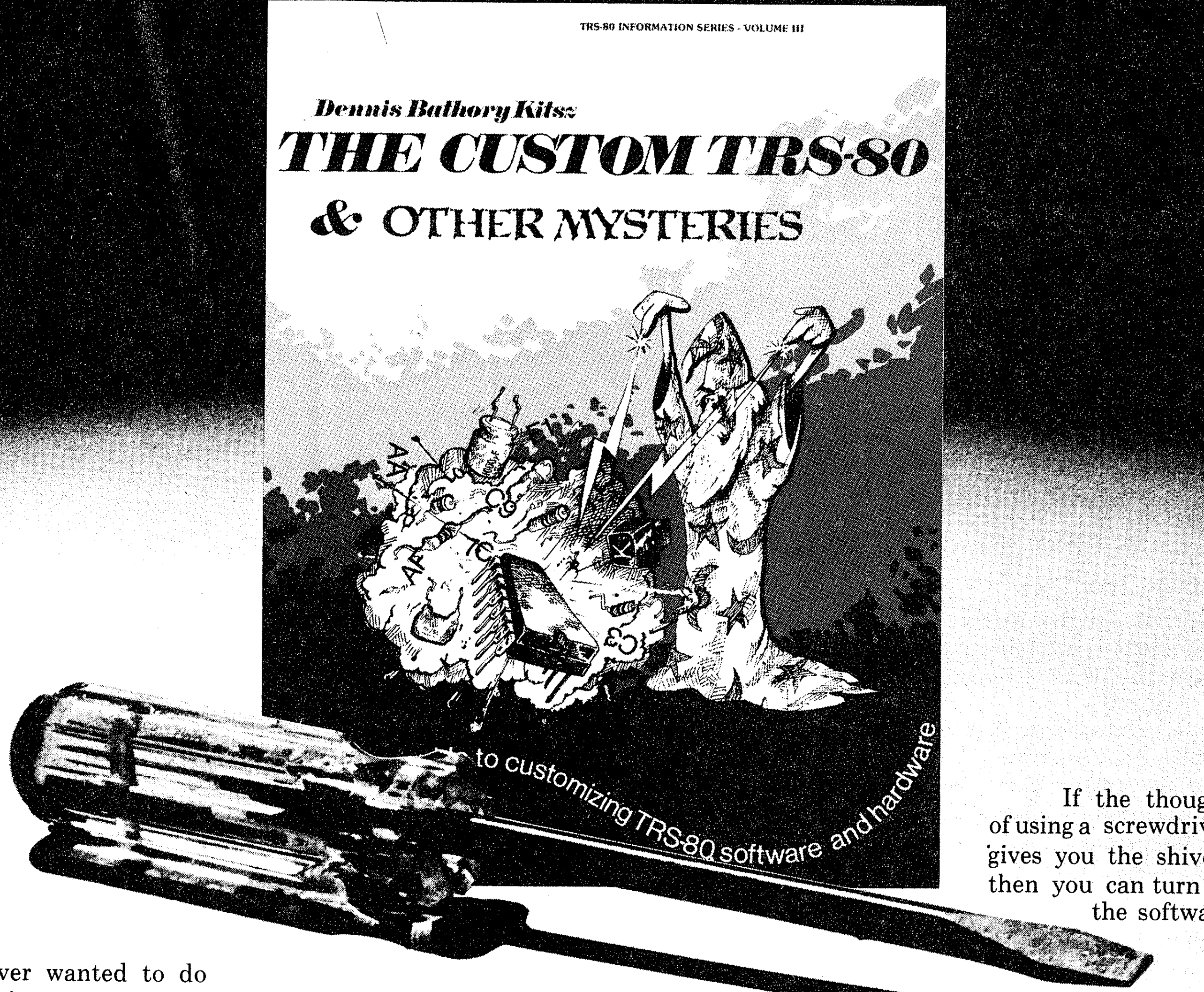
If IJG products are not currently carried by your local dealer, order direct from IJG.

Include $4.00 for shipping and handling. Foreign residents add $11.00 plus purchase price. U.S. funds only please.

IJG, Inc. 1953 W 11th Street
Upland, California 91786
Phone: 714/946-5805

Eighteen BASIC Adventures & Program Generator for the TRS-80 Model I & III. Accept the Challenge.

TUNE-UP YOUR TRS-80


TRS-80 INFORMATION SERIES - VOLUME III
Dennis Bathory Kitsz
THE CUSTOM TRS-80
& OTHER MYSTERIES
to customizing TRS-80 software and hardware

If the thought of using a screwdriver gives you the shivers then you can turn to the software section.

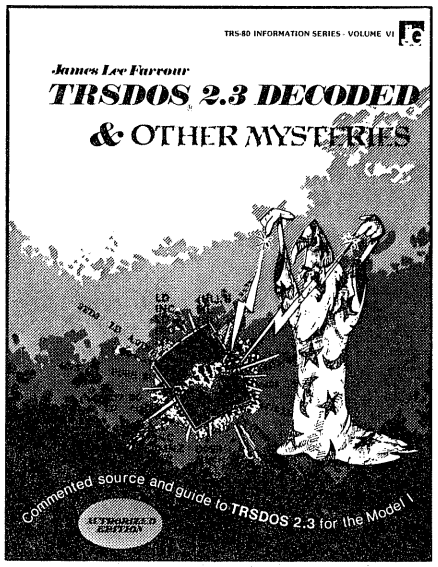
In this you learn how to make BASIC programs auto-execute, reset the memory size, patch into the interpreter, test memory with machine-language, pack program lines with machine code, and generate sound effects.

Ever wanted to do things to your TRS-80 that Radio Shack said couldn't be done? How about upper/lower case, reverse video, high-resolution graphics, a high-speed clock, audible keystrokes, an extra keyboard, and a real-time clock? Still not enough? How about using an 8-track as a mass storage device, making music, controlling a synthesiser, monitoring your data bus, and individual reverse characters? All these hardware modifications, plus lots more, are in **The Custom TRS-80** and Other Mysteries, by **Dennis Bathory Kitsz** – the latest book from **IJG Computer Services.**

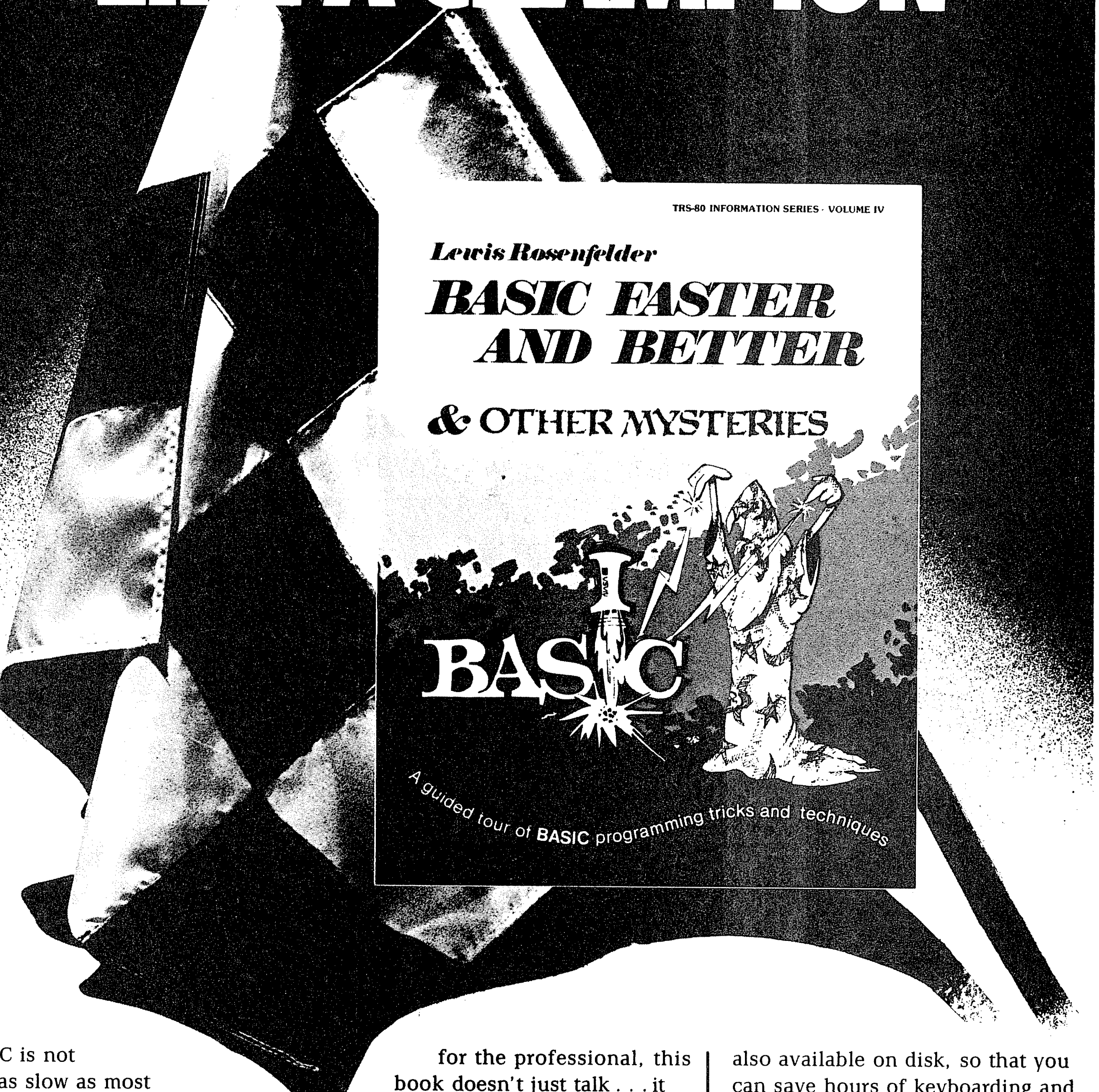
The Custom TRS-80 and Other Mysteries is more than 300 pages of practical information, and tested software, for **$29.95**. Order your copy now, and start turning your TRS-80 into a five-hundred-dollar supercomputer! Pick one up at your local **IJG** dealer or use the coupon on the opposite page.

IJG

TRS-80 is a trademark of Tandy


TRS-80 INFORMATION SERIES · VOLUME VI
James Lee Farvour
TRSDOS 2.3 DECODED & OTHER MYSTERIES
Commented source and guide to TRSDOS 2.3 for the Model I
AUTHORIZED EDITION

MAKE BASIC PERFORM LIKE A CHAMPION


TRS-80 INFORMATION SERIES - VOLUME IV
Lewis Rosenfelder
BASIC FASTER AND BETTER
& OTHER MYSTERIES
BASIC
A guided tour of BASIC programming tricks and techniques

BASIC is not nearly as slow as most programmers think it is. *BASIC Faster and Better* shows you how to super-charge your BASIC, with 300 pages of fast, functions and subroutines.

You won't find any trivial 'check-book-balancing' programs in this book – it's packed with *useful* programs.

Tutorial for the beginner, instructive for the advanced, and invaluable for the professional, this book doesn't just talk . . . it shows how! All of the routines are also available on disk, so that you can save hours of keyboarding and debugging.

BASIC Faster & Better and Other Mysteries is $29.95, and the two program disks are $19.95 each. The #1 disk *BFBDEM* contains all the demonstration programs, and #2 *BFBLIB* has all the library functions.

Get the book and/or disks from your local *IJG* dealer and B. Dalton bookstores.

IJG

1953 West
11th Street
Upland, CA
91786 (714)
946-5805

ISBN 0-936200-08-1