August 1991

REMark

A PC Video Primer
Page 5

Melodies
For Your Programs
Page 36

Watch for New
Zenith Data Systems Products
Coming in Future Issues

The Official Zenith Data Systems Magazine

Share the Knowledge!

Have you done something interesting with your computer lately? Found a piece of software or hardware you don't know how you got along without? Designed a new product, be it software or hardware, for your system? By submitting this information in the form of a major article, you can share with others the knowledge of a particular subject. REMark magazine is currently looking for authors (novice or professional) to write articles. Even if you have never written before, give it a try! As a REMark author, you will recieve up to $400 for each article accepted and published. (For more information on current policies, call Lori Lerch at 616-982-3794.)

So... Let's get to it and
Share the Knowledge!

REMark®

August 1991

The Official Zenith Data Systems Users Magazine

Managing Editor
Jim Buszkiewicz
(616) 982-3837

Software Engineer
Pat Swayne
(616) 982-3463

Production Coordinator
Lori Lerch
(616) 982-3794

Secretary
Lisa Cobb
(616) 982-3463

COM1 Bulletin Board
(616) 982-3956
(Modem Only)

ZUG Software Orders
(616) 982-3463

Contributing Editor
William M. Adney

Printer
Imperial Printing
St. Joseph, MI

Contributing Editor
Robert C. Brenner

Advertising
Rupley's Advertising Service
Dept. REM, 240 Ward Avenue
P.O. Box 348
St. Joseph, MI 49085-0348
(616) 983-4550

To Locate your Nearest:
Dealer 1-800-523-9393
Service Center 1-800-777-4630

	U.S. Domestic	**APO/FPO & All Others**
Initial	$22.95	$37.95*
Renewal	$19.95	$32.95*

* U.S. Funds

Limited back issues are available at $2.50, plus 10% shipping and handling - minimum $1.00 charge. Check ZUG Product List for availability of bound volumes of past issues. Requests for magazines mailed to foreign countries should specify mailing method and include appropriate, additional cost.

Send Payment to: Zenith Users' Group
P.O. Box 217
Benton Harbor, MI 49023-0217
(616) 982-3463

Software

PRODUCT NAME	PART NUMBER	OPERATING SYSTEM	DESCRIPTION	PRICE
	H8 - H/Z-89/90			
ACTION GAMES	885-1220-[37]	CPM	GAME	20.00
ADVENTURE	885-1010	HDOS	GAME	10.00
ASCIRITY	885-1238-[37]	CPM	AMATEUR RADIO	20.00
AUTOFILE (Z80 ONLY)	885-1110	HDOS	DBMS	30.00
BHBASIC SUPPORT PKG	885-1119-[37]	HDOS	UTILITY	20.00
CASTLE	885-8032-[37]	HDOS	ENTERTAINMENT	20.00
CHEAPCALC	885-1131-[37]	HDOS	SPREADSHEET	20.00
CHECKOFF	885-8010	HDOS	CHKBK SOFTWARE	25.00
DEVICE DRIVERS	885-1105	HDOS	UTILITY	20.00
DISK UTILITIES	885-1213-[37]	CPM	UTILITY	20.00
DUNGEONS & DRAGONS	885-1093-[37]	HDOS	GAME	20.00
FLOATING POINT PKG	885-1063	HDOS	UTILITY	18.00
GALACTIC WARRIORS	885-8009-[37]	HDOS	GAME	20.00
GALACTIC WARRIORS	885-8009-[37]	CPM	GAME	20.00
GAMES 1	885-1029-[37]	HDOS	GAMES	18.00
HARD SECT SUPPORT PKG	885-1121	HDOS	UTILITY	30.00
HDOS PROG. HELPER	885-8017	HDOS	UTILITY	16.00
HOME FINANCE	885-1070	HDOS	BUSINESS	18.00
HUG DISK DUP UTILITY	885-1217-[37]	CPM	UTILITY	20.00
HUG SOFTWARE CATALOG	885-4500	VARIOUS	PROD TO 1982	9.75
HUGMAN & MOVIE ANIM	885-1124	HDOS	ENTERTAINMENT	20.00
INFO SYS AND TEL. & MAIL SYS	885-1108-[37]	HDOS	DBMS	30.00
LOGBOOK	885-1107-[37]	HDOS	AMATEUR RADIO	30.00
MAGBASE	885-1249-[37]	CPM	MAGAZINE DB	25.00
MISCELLANEOUS UTILITIES	885-1089-[37]	HDOS	UTILITY	20.00
MORSE CODE TRANSCEIVER	885-8016	HDOS	AMATEUR RADIO	20.00
MORSE CODE TRANSCEIVER	885-8031-[37]	CPM	AMATEUR RADIO	20.00
PAGE EDITOR	885-1079-[37]	HDOS	UTILITY	25.00
PROGRAMS FOR PRINTERS	885-1082	HDOS	UTILITY	20.00
REMARK VOL 1 ISSUES 1-13	885-4001	N/A	1978 TO DEC '80	20.00
RUNOFF	885-1025	HDOS	TEXT PROCR	35.00
SCICALC	885-8027	HDOS	UTILITY	20.00
SMALL BUISNESS PACKAGE	885-1071-[37]	HDOS	BUSINESS	75.00
SMALL-C COMPILER	885-1134	HDOS	LANGUAGE	30.00
SOFT SECTOR SUPPORT PKG	885-1127-[37]	HDOS	UTILITY	20.00
STUDENT'S STATISTICS PKG	885-8021	HDOS	EDUCATION	20.00
SUBMIT (Z80 ONLY)	885-8006	HDOS	UTILITY	20.00
TERM & HTOC	885-1207-[37]	CPM	COMMUN & UTIL	20.00
TINY BASIC COMPILER	885-1132-[37]	HDOS	LANGUAGE	25.00
TINY PASCAL	885-1086-[37]	HDOS	LANGUAGE	20.00
UDUMP	885-8004	HDOS	UTILITY	35.00
UTILITIES	885-1212-[37]	CPM	UTILITY	20.00
UTILITIES BY PS	885-1126	HDOS	UTILITY	20.00
VARIETY PACKAGE	885-1135-[37]	HDOS	UTILITY & GAMES	20.00
WHEW UTILITIES	885-1120-[37]	HDOS	UTILITY	20.00
XMET ROBOT X-ASSEMBLER	885-1229-[37]	CPM	UTILITY	20.00
Z80 ASSEMBLER	885-1078-[37]	HDOS	UTILITY	25.00
Z80 DEBUGGING TOOL (ALDT)	885-1116	HDOS	UTILITY	20.00
	H8 - H/Z-89/90 - H/Z-100 (Not PC)			
ADVENTURE	885-1222-[37]	CPM	GAME	10.00
BASIC-E	885-1215-[37]	CPM	LANGUAGE	20.00
CASSINO GAMES	885-1227-[37]	CPM	GAME	20.00
CHEAPCALC	885-1233-[37]	CPM	SPREADSHEET	20.00
CHECKOFF	885-8011-[37]	CPM	CHKBK SOFTWARE	25.00
COPYDOS	885-1235-[37]	CPM	UTILITY	20.00
DISK DUMP & EDIT UTILITY	885-1225-[37]	CPM	UTILITY	30.00
DUNGEONS & DRAGONS	885-1209-[37]	CPM	GAMES	20.00
FAST ACTION GAMES	885-1228-[37]	CPM	GAME	20.00
FUN DISK I	885-1236-[37]	CPM	GAMES	20.00
FUN DISK II	885-1248-[37]	CPM	GAMES	35.00
GAMES DISK	885-1206-[37]	CPM	GAMES	20.00
GRADE	885-8036-[37]	CPM	GRADE BOOK	20.00
HRUN	885-1223-[37]	CPM	HDOS EMULATOR	40.00
HUG FILE MANAGER & UTILITIES	885-1246-[37]	CPM	UTILITY	20.00
HUG SOFTWARE CAT UPDT #1	885-4501	VARIOUS	PROD 1983 TO 1985	9.75
KEYMAP CPM-80	885-1230-[37]	CPM	UTILITY	20.00
MBASIC PAYROLL	885-1218-[37]	CPM	BUSINESS	60.00
NAVPROGSEVEN	885-1219-[37]	CPM	FLIGHT UTILITY	20.00
SEA BATTLE	885-1211-[37]	CPM	GAME	20.00
UTILITIES BY PS	885-1226-[37]	CPM	UTILITY	20.00
UTILITIES	885-1237-[37]	CPM	UTILITY	20.00
X-REFERENCE UTIL FOR MBASIC	885-1231-[37]	CPM	UTILITY	20.00
ZTERM	885-3003-[37]	CPM	COMMUNICATIONS	20.00

Price List

PRODUCT NAME	PART NUMBER	OPERATING SYSTEM	DESCRIPTION	PRICE
H/Z-100 (Not PC) Only				
CARDCAT	885-3021-37	MSDOS	BUSINESS	20.00
CHEAPCALC	885-3006-37	MSDOS	UTILITY	20.00
CHECKBOOK MANAGER	885-3013-37	MSDOS	BUSINESS	20.00
CP/EMULATOR	885-3007-37	MSDOS	CPM EMULATOR	20.00
DBZ	885-8034-37	MSDOS	DBMS	25.00
DUNGN & DRAGONS (ZBASIC)	885-3009-37	MSDOS	GAME	20.00
ETCHDUMP	885-3005-37	MSDOS	UTILITY	20.00
EZPLOT II	885-3049-37	MSDOS	PRINTER PLOT UTIL	25.00
GAMES (ZBASIC)	885-3011-37	MSDOS	GAMES	20.00
GAMES CONTEST PACKAGE	885-3017-37	MSDOS	GAMES	25.00
GAMES PACKAGE II	885-3044-37	MSDOS	GAMES	25.00
GRAPHIC GAMES (ZBASIC)	885-3004-37	MSDOS	GAMES	20.00
GRAPHICS	885-3031-37	MSDOS	UTILITY	20.00
HELPSCREEN	885-3039-37	MSDOS	UTILITY	20.00
HUG BKGRD PRINT SPOOLER	885-1247-37	CPM	UTILITY	20.00
KEYMAC	885-3046-37	MSDOS	UTILITY	20.00
KEYMAP	885-3010-37	MSDOS	UTILITY	20.00
KEYMAP CPM-85	885-1245-37	CPM	UTILITY	20.00
MATHFLASH	885-8030-37	MSDOS	EDUCATION	20.00
ORBITS	885-8041-37	MSDOS	EDUCATION	25.00
POKER PARTY	885-8042-37	MSDOS	ENTERTAINMENT	20.00
SCICALC	885-8028-37	MSDOS	UTILITY	20.00
SKYVIEWS	885-3015-37	MSDOS	ATRONOMY UTILITY	20.00
SMALL-C COMPILER	885-3026-37	MSDOS	LANGUAGE	30.00
SPELL5	885-3035-37	MSDOS	SPELLING CHECKER	20.00
SPREADSHEET CONTEST PKG	885-3018-37	MSDOS	VARIOUS SPRDST	25.00
TREE-ID	885-3036-37	MSDOS	TREE IDENTIFIER	20.00
USEFUL PROGRAMS I	885-3022-37	MSDOS	UTILITIES	30.00
UTILITIES	885-3008-37	MSDOS	UTILITY	20.00
ZPC II	885-3037-37	MSDOS	PC EMULATOR	60.00
ZPC UPGRADE DISK	885-3042-37	MSDOS	UTILITY	20.00
H/Z-100 and PC Compatibles				
ADVENTURE	885-3016	MSDOS	GAME	10.00
BACKGRD PRINT SPOOLER	885-3029	MSDOS	UTILITY	20.00
BOTH SIDES PRINTER UTILITY	885-3048	MSDOS	UTILITY	20.00
CXREF	885-3051	MSDOS	UTILITY	17.00
DEBUG SUPPORT UTILITIES	885-3038	MSDOS	UTILITY	20.00
DPATH	885-8039	MSDOS	UTILITY	20.00
HADES II	885-3040	MSDOS	UTILITY	40.00
HEPCAT	885-3045	MSDOS	UTILITY	35.00
HUG EDITOR	885-3012	MSDOS	TEXT PROCESSOR	20.00
HUG MENU SYSTEM	885-3020	MSDOS	UTILITY	20.00
HUG SOFTWARE CAT UPD #1	885-4501	MSDOS	PROD 1983 - 1985	9.75
HUGMCP	885-3033	MSDOS	COMMUNICATION	40.00
ICT 8080 - 8088 TRANSLATOR	885-3024	MSDOS	UTILITY	20.00
MAGBASE	885-3050	VARIOUS	MAG DATABASE	25.00
MATT	885-8045	MSDOS	MATRIX UTILITY	20.00
MISCELLANEOUS UTILITIES	885-3025	MSDOS	UTILITIES	20.00
PS' PC &Z100 UTILITIES	885-3052	MSDOS	UTILITIES	20.00
REMARK VOL 8 ISSUES 84-95	885-4008	N/A	1987	25.00
REMARK VOL 9 ISSUES 96-107	885-4009	N/A	1988	25.00
REMARK VOL 10 ISSUES 108-119	885-4010	N/A	1989	25.00
REMARK VOL 11 ISSUES 120-131	885-4011	N/A	1990	25.00
SCREEN DUMP	885-3043	MSDOS	UTILITY	30.00
UTILITIES II	885-3014	MSDOS	UTILITY	20.00
Z100 WORDSTAR CONNECTION	885-3047	MSDOS	UTILITY	20.00
PC Compatibles				
CARDCAT	885-6006	MSDOS	CAT SYSTEM	20.00
CHEAPCALC	885-6004	MSDOS	SPREADSHEET	20.00
CLAVIER	885-6016	MSDOS	ENTERTAINMENT	20.00
CP/EMULATOR II & ZEMULATOR	885-6002	MSDOS	CPM & Z100 EMUL	20.00
DUNGEONS & DRAGONS	885-6007	MSDOS	GAME	20.00
EZPLOT II	885-6013	MSDOS	PRINTER PLOT UTIL	25.00
GRADE	885-8037	MSDOS	GRADE BOOK	20.00
HAM HELP	885-6010	MSDOS	AMATEUR RADIO	20.00
KEYMAP	885-6001	MSDOS	UTILITY	20.00
LAPTOP UTILITIES	885-6014	MSDOS	UTILITIES	20.00
PS' PC UTILITIES	885-6011	MSDOS	UTILITIES	20.00
POWERING UP	885-4604	N/A	GUIDE TO USING PCs	12.00
SCREEN SAVER PLUS	885-6009	MSDOS	UTILITIES	20.00
SKYVIEWS	885-6005	MSDOS	ASTRONOMY UTIL	20.00
TCSPELL	885-8044	MSDOS	SPELLING CHECKER	20.00
ULTRA RTTY	885-6012	MSDOS	AMATEUR RADIO	20.00
YAUD (YET ANOTHER UTIL DSK)	885-6015	MSDOS	UTILITIES	20.00

This Software Price List contains all products available for sale. For a detailed abstract of these products, refer to the Software Catalog, Software Catalog Update #1, or previous issues of REMark.

Now Available!
ZUG software is now available on 2" disks. Just put a "-90" at the end of the part number (i.e., 885-6014-90). Also add $3.00 to the purchase price of the software (i.e., $20.00 + $3.00 = $23.00).

LAPTOP OWNERS . . . don't feel left out! All of ZUG's MS-DOS software is available on 3-1/2" micro-floppies too! When ordering, just add a "-80" to the 7-digit ZUG part number. For the standard 5-1/4" floppy, just add a "-37".

Make the no-hassle connection with your modem today! HUGMCP doesn't give you long menus to sift through like some modem packages do. With HUGMCP, YOU'RE always in control, not the software. Order HUG P/N 885-3033-37 today, and see if it isn't the easiest-to-use modem software available. They say it's so easy to use, they didn't even need to look at the manual. "It's the only modem software that I use, and I'm in charge of the HUG bulletin board!" says Jim Buszkiewicz. HUGMCP runs on ANY Heath/Zenith computer that's capable of running MS-DOS!

ORDERING INFORMATION

For VISA, MasterCard, and American Express phone orders, telephone the Zenith Users' Group directly at (616) 982-3463. Have the part number(s), description(s), and quantity ready for quick processing. By mail, send your order, plus 10% postage and handling ($1.00 minimum charge, up to a maximum of $5.00) to: Zenith Users' Group, P.O. Box 217, Benton Harbor, MI 49023-0217. VISA, MasterCard and American Express require minimum $10.00 order. No C.O.D.s accepted.

Questions regarding your subscription? Call Lisa Cobb at (616) 982-3463.

ZENITH
data systems
Groupe Bull

REMark Magazine Subscription & ZLink/COM1 Bulletin Board Information

Your subscription entitles you to receive REMark, our monthly magazine containing articles specific to Zenith Data Systems computer and generally to other PC Compatible computers. All articles in REMark are submitted by readers like you. We welcome YOUR articles, and will pay you for any we accept!

A REMark subscription also allows you full access to the ZLink-COM1 bulletin board system (COM1, for short) described in detail in the brochure. There are many, many megabytes of free and shareware software available for downloading to registered COM1 users. Full access also lets you order products from the "Bargain Centre" section of COM1. The money you can save in the Keyboard Shopping Club will pay for decades of REMark subscriptions.

Last, but definitely not least, your subscription puts you in touch with thousands of other Zenith Data System computer users, from whom invaluable information can be exchanged.

REMark subscriptions, currently $22.95, can be obtained in one of three ways. First, by ordering one on the COM1 bulletin board (see the Keyboard Shopping Club section); second, by phone with VISA, MasterCard, or American Express; and third, through the US Mail using a credit card, money order or check made payable to: Zenith Data Systems. Our address is:

Zenith Data Systems Users' Group
P.O. Box 217
Benton Harbor, MI 49023-0217
(616) 982-3463

Once you receive your ID number, registration on the COM1 BBS is NOT automatic. It requires that you log on, enter your first name and last name EXACTLY as they appear on your REMark mailing label, and then enter your ID number as your password. The FIRST time you access the board, you must elect to start a NEW ACCOUNT and answer the various questions. Once you've done this, our automated scanner will compare the system's database against the subscription database. If you made no mistakes, you will be verified and given full access, within 24 hours.

Once you've been authorized as a full member, several important things happen. First, you're given full downloading privileges of up to one megabyte per day. Secondly, you'll have full access to the message boards. And finally, you'll be able to take full advantage of the Bargain Centre product savings.

Detach this form, enclose your check, money order or credit card information (no cash please).

REMark Subscription / Renewal Form

New Member: ☐ Yes ☐ No Credit Card # ____________________

ID Number: ____________________ Exp. Date ____________________

Address Change? ☐

		Renew	New
Name: ____________________	U.S. Bulk Mail	☐ 19.95	☐ 22.95
Address: ____________________	U.S. First Class	☐ 32.95	☐ 37.95
City, State, Zip: ____________________	APO/FPO Surface Overseas	☐ 32.95	☐ 37.95
Daytime Phone #: ( ) ____________________	Air Printed Overseas	☐ 52.95	☐ 57.95

A PC Video Primer

David R. Veit
Technical Writer
Zenith Data Systems

HISTORY

When the original IBM PC was introduced in 1981, it was equipped with a video subsystem that was capable of displaying hi-resolution monochrome text only. For those first PC users in the dawn of the PC era, it was more than adequate. But over the next ten years as PC hardware and application software have undergone vast improvement, users have demanded more from their machines, and video quality was no exception. During this time, a variety of video adapters have been introduced, all designed to satisfy PC user's growing demands. Some have gone on to become industry standards, while a flurry of others have failed to gain popular acceptance and have either disappeared entirely or serve as the basis for proprietary display systems.

It was during this first decade that the PC was undergoing a process of vast acceptance by the computing public, and consequently, the PC industry was experiencing tremendous growth. Part of the reason all of this was happening was that the industry had embraced the original IBM PC architecture and had accepted it as a defacto standard. Subsequent IBM machines, the PC/XT and PC/AT, were also embraced and accepted as "standard". The video solutions within these IBM machines survived and have become part of the accepted PC architecture that is followed to this day. The IBM video solutions that have become widely accepted standards are as follows:

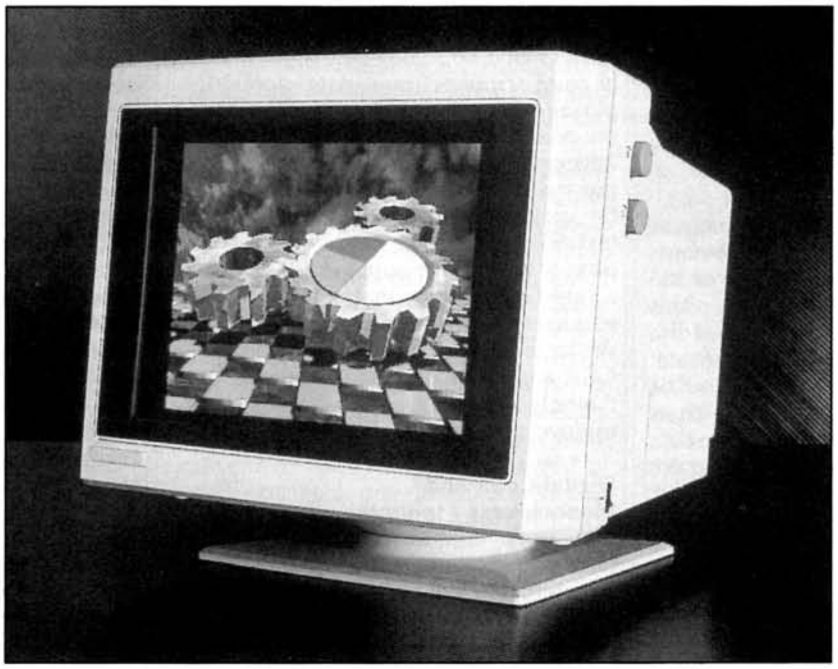
MDA	-	**M**onochrome **D**isplay **A**dapter
CGA	-	**C**olor **G**raphics **A**dapter
EGA	-	**E**nhanced **G**raphics **A**dapter
VGA	-	**V**ideo **G**raphics **A**rray

Similar to the MDA architecture, but with additional graphics capabilities that the IBM did not have, and widely accepted by industry is the:

HGC - **H**ercules **G**raphics **C**ard

IBM has introduced other video adapters, including the Professional Graphics Controller (PGC) and the 8514/A hi-resolution adapter, but these did not gain wide acceptance in the industry. The Extended Graphics Adapter (XGA) is now a newcomer to the video arena, but at the moment, it is confined for use in IBM's PS/2 MicroChannel Architecture machines only.

There are other on-going attempts at standardizing video architectures, but all of them fall short in that they cannot replace the original architecture as originally defined by IBM. At present, there is no getting around the fact that all IBM-compatible Industry-Standard Architecture (ISA) PCs must have an IBM-compatible video adapter present in the machine. Other video architectures that have strong backing these days are the Texas Instruments Graphics Architecture (TIGA) and a VGA extension called SuperVGA as proposed by the Video Electronics Standards Association (VESA), a consortium of companies organized for the purpose of promoting standardization within the PC video industry.



MONITOR BASICS

The display monitors used in today's computer systems are specialized devices capable of displaying digital data information that is stored in a video buffer. The computer's video display adapter is used as a source to drive the display by providing data from the video data buffer and all control signals required by the monitor.

Resolution

One of the most important qualities of a monitor is the degree of resolution that information can be displayed on the screen. Resolution tells us how many discrete picture elements (commonly called pixels or pels) are to be resolved by the monitor. For example, all VGA analog monitors are capable of displaying data at a 640H x 480V format. This means each line of information is 640 pixels wide (horizontal resolution), and the monitor is displaying 480 lines (vertical resolution), with each line one pixel high. The monitor can display lines of information only as fast as its line rate. This line rate is also known as the horizontal refresh rate because it is the rate at which the electron gun(s) will stop and retrace horizontally from the far right edge of the screen back to the far left edge of the screen in preparation for the next line. The monitor will also refresh the screen at some particular vertical refresh rate, which is the rate at which the monitor will retrace vertically from the lower right corner of the display up to the upper left corner. This is also known as the frame refresh rate.

Vertical Resolution

A monitor's vertical resolution is the number of lines that will be seen on the screen. The maximum theoretical number of lines can be calculated by dividing the horizontal refresh rate (lines per second) by the vertical refresh rate (display frames per second). For a typical analog monitor used in VGA systems:

```
31.47 kHz (lines/sec.)
------------------------- = 52 lines/ frame
60 Hz     (frames/sec.)
```

The VGA analog monitor utilizes a 60 Hz frame rate only in modes requiring a 640x480 resolution. While displaying 525 lines seems possible in the above calculation, that could only be possible if the monitor could display all 525 lines and then perform a vertical retrace to start again at the top of the screen without any time elapsing. Instead, we can display only 480 lines because the time it would take to display the other 45 lines (i.e., 525 - 480) is used to perform the monitor's vertical retrace operation.

Horizontal Resolution

A monitor's horizontal resolution is the number of discrete picture elements that can be generated across each of the display lines. The rate at which these pixels are displayed on the screen is called the pixel rate, sometimes called the dot rate. This rate must be very accurate and is usually controlled by a crystal oscillator. The theoretical maximum number of pixels per line can be calculated by dividing the pixel rate by the line rate. For a typical VGA analog monitor system:

```
25.175 MHz (pixels/sec.)
------------------------- = 800 pixels/line
31.47 kHz  (lines/sec.)
```

Like the calculation for vertical resolution, this 800 pixels/line maximum is really impossible. The typical VGA mode will have 640 pixels per line, with the time allotted for the remaining 160 pixels (i.e., 800 - 640) used for the monitor's horizontal retrace operations.

Monitor Types

There are several types of computer monitors in use today, and they are generally classified by the type of input signal accepted. Monitors can be classifed as color or monochrome, and will fall into one of three major categories: composite, digital, and analog.

COLOR MONITORS

Color monitors contain three electron guns, one for each of the three primary colors: red, green, and blue. The face of the tube is also coated with special phosphors which are arranged in a repeatable pattern of red, green, and blue phosphors. Each of these electron guns are then controlled in such a fashion as to send an electron stream at the phosphor groups on the face, causing them to glow and emit color. The red gun will excite the red phosphors, the green gun the green phosphors, and similarly the blue gun will excite the blue phosphors.

Monochrome Monitors

Monochrome monitors operate similar to the color monitors except that they contain only one electron gun. The face of the tube is also coated with a single color phosphor material, instead of groups of red, green, and blue phosphors. The electron gun will send an electron stream at the face and excite the phosphors, just like in the color monitor. Common phosphor colors for monochrome monitors are white, amber, and green.

Composite Monitors

Composite monitors accept a composite signal, an analog-type signal that has all color or monochrome data information, plus all control signal information encoded on one signal line. Composite monitors have only two lines: one signal line and one ground line. These monitors are still in use, but very few are sold new these days. The technology is simply outdated. The purpose of having composite video output on a video card was so that one could use a very inexpensive monitor, with electronics similar to a television set. The original IBM CGA and compatible video adapters are the only adapters that have the capability to drive a composite monitor.

Digital Color and Monochrome Monitors

Digital monitors, also called direct-drive monitors, accept individual TTL-level signals for each of the required colors. The number of colors that a digital monitor can display is dependent on the number of input color lines. For example, the Color Display used in Color Graphics Adapter (CGA) systems has one line for each of the colors: red, green, blue, plus one for intensity. These four lines are capable of displaying colors from a palette of 16 color combinations (i.e., $2^4 = 16$). Likewise, a monochrome monitor has only a single color input line and needs only to be on or off to display its two colors.

Analog Color and Monochrome Monitors

Analog monitors came on to the scene when higher color depth and resolution were required. It has three analog input color lines, one each for red, blue, and green. The color displayed is determined by the voltage applied to each of the color lines, not simply by the "on" or "off" state of the color lines as in digital monitors. For example, if the voltage on any particular color line increases, the intensity of that color will likewise increase. The number of colors an analog monitor can potentially display is practically infinite, because each of the color input lines can have its voltage level (i.e, intensity) varied to some level to achieve a desired color. The primary advantage of this type of setup is that virtually any color combination is possible by varying the signal strength of each of these color lines. In reality, things are a little different. Typically, a 6-bit digital-to-analog converter (DAC) will control the voltage level for each of the three primary color lines. So, instead of an infinite number of possible intensity levels for each of the lines, the DACs will limit each color to one of 64 discrete levels ($2^6 = 64$). With three DACs, the system is then capable of selecting colors from a palette of 262,144 colors (i.e, $2^{(3*6)} = 262,144 = 256K$). Monochrome systems are similar except that they have only one "color". Its six-bit DAC would therefore generate 64 (i.e., $2^6 = 64$) "colors", seen as shades of gray on a mono-

Adapter	Pixel Clock	Horiz. Refresh	Vert. Refresh	Monitor Type	Resolution
MDA	16.257 MHz	18.43 KHz	50 Hz	MD, ECD	720x350
HGC	16.257 MHz	18.43 KHz	50 Hz	MD, ECD	720x350
CGA	14.318 MHz	15.75 KHz	60 Hz	CD, ECD	640x200
EGA	16.257 MHz	21.85 KHz	60 Hz	ECD	640x350
	14.318 MHz	15.75 KHz	60 Hz	CD	640x200
	16.257 MHz	18.43 KHz	50 Hz	MD	720x350
VGA	25.175 MHz	31.47 KHz	70 Hz	Analog	640x350
	25.175 MHz	31.47 KHz	70 Hz	Analog	640x400
	25.175 MHz	31.47 KHz	60 Hz	Analog	640x480
	25.175 MHz	31.47 KHz	70 Hz	Analog	720x400

Monitor legend:		
	MD	- IBM Monochrome Display or equivalent
	CD	- IBM Color Display or equivalent
	ECD	- IBM Enhanced Color Display or equivalent
	Analog	- IBM VGA Analog display or equivalent

Figure 1

chrome monitor.

Interlaced and Non-Interlaced Monitors

When IBM introduced its 8514/A hi-resolution graphics adapter in 1987, it required use of an "interlaced" monitor. It is the only video adapter in common use today that requires use of an interlaced display.

Most computer monitors will display all horizontal scan lines every time the entire display frame is refreshed. For example, if there are 768 lines of resolution, the monitor will be driven to display all 768 lines before a frame refresh. This is normal, "non-interlaced" operation. Interlaced monitors take another approach by alternately displaying all even-numbered scan lines on one frame refresh, and then all odd-numbered scan lines on the following frame refresh. This interlacing technique requires two frame refreshes to get all of the information to the screen instead of the usual one refresh. As in our example above, even-numbered lines 0,2,4,...764,766 are displayed during one frame, and then odd-numbered lines 1,3,5,...765,767 are displayed on the following frame. A higher resolution requires a much higher line rate, but a smaller line rate can be used if an interlacing technique is utilized. This technique is also used to compensate for hardware that is simply inadequate (read: cheap) for the task it is required to perform. Interlaced displays generally have a slightly higher frame refresh rate than non-interlaced displays in order to keep the resultant display flicker to a tolerable minimum. Another technique used to reduce display flicker in interlaced displays is for the display to utilize long-persistence phosphors that will glow for a slightly longer period of time than "normal" medium-persistence phosphors when excited by the electron beam.

IBM-Compatible Display Monitors

There are several types of "standard" monitors that are used with IBM video adapters. Each of these monitors were introduced by IBM and were designed to operate with a specified adapter (See Figure 1).

ADAPTER BASICS

All popular display adapters for PC-compatible computers are considered "dumb" adapters because they have no on-board processing capabilities. All video-related operations are handled by the system processor. The MDA, Hercules, CGA, EGA, VGA, and SuperVGA adapters are in this "dumb-adapter" class. There are other display adapters available that do have on-board processing capability, but these are limited to some of the newer adapters that provide extra horsepower for handling higher color and pixel resolutions, and graphic intensive applications such as Microsoft Windows 3.0 and CAD packages. This would include adapters based on Texas Instruments' 34010 and 34020 graphics processors or TIGA software interface, and IBM's 8514/A and XGA adapters.

The primary purpose of any PC-compatible video adapter is to store, manipulate, and send data to the display. The manner in which it does these tasks will differ among adapter types.

Alphanumeric Data Format

All PC display adapters handle text operation in a similar manner. A video data buffer, usually implemented with a block of dual-ported video RAM, is used to store information on each character to be displayed on the screen. Each character to be displayed uses two bytes of video memory to describe it. One byte describes the actual character with an ASCII code (00-FFh). A second byte describes the actual character's attributes. Depending on the adapter type, these attributes are intensity, blinking, and normal or reverse video.

To get characters onto the screen, the adapter's CRT controller circuitry reads the contents of a pair of video memory locations: one is the character's ASCII code, the other its attributes. The ASCII code is then translated by an on-board *character generator* containing the dot patterns necessary to generate each character. These patterns are sent off to a shift register where they can be fed to the video display one dot at a time. On the way out, a special video logic circuit can modify the dot stream according to the character's desired attributes.

The actual size of the characters seen on the screen differs among the adapter types. Since different adapters have different display resolution capabilities, one can quickly calculate the size of the "cell" or "block" reserved for each character's dot pattern. For example, the VGA utilizes a 720 x 400 pixel resolution when in its default text mode. The resultant 80 column x 25 lines of text indicate that each cell is 9 pixels wide and 16 pixels high (9 x 16 cell).

MDA video mode:

Video Mode	Type	Resolution & Colors	Text Scheme	Char Cell	# Display Pages	Video Buffer
7	text	720 x 350 mono	80 x 25	9x14	8	B8000h

CGA video modes:

Mode	Type	Resolution	Colors	Text Scheme	Char Cell	# Display Pages	Video Buffer
0	text	320x200	16	40 x 25	8x8	8	B8000h
1	text	320x200	16	40 x 25	8x8	8	B8000h
2	text	640x200	16	80 x 25	8x8	8	B8000h
3	text	640x200	16	80 x25	8x8	8	B8000h
4	APA	320x200	4	40 x 25	8x8	1	B8000h
5	APA	320x200	4	40 x 25	8x8	1	B8000h
6	APA	640x200	2	80 x 25	8x8	1	B8000h

EGA video modes:

Mode	Type	Displ.	Resolution	Colors	Text Scheme	Char Cell	Display Pages	Video Buffer
0	text	CD	320x200	16	40x25	8x8	8	B8000h
0	text	ECD	320x350	16	40x25	8x14	8	B8000h
1	text	CD	320x200	16	40x25	8x8	8	B8000h
1	text	ECD	320x350	16	40x25	8x14	8	B8000h

Figure 2 (Cont'd.)

2	text	CD	640x200	16	80x25	8x8	8	B8000h
2	text	ECD	640x200	16	80x25	8x14	8	B8000h
3	text	CD	640x200	16	80x25	8x8	8	B8000h
3	text	ECD	640x200	16	80x25	8x14	8	B8000h
4	APA	CD/ECD	320x200	4	40x25	8x8	1	B8000h
5	APA	CD/ECD	320x200	4	40x25	8x8	1	B8000h
6	APA	CD/ECD	640x200	2	80x25	8x8	1	B8000h
7	text	MD	720x350	Mono	80x25	9x14	8	B0000h
D	APA	CD/ECD	320x200	16	40x25	8x8	2/4/8	A0000h
E	APA	CD/ECD	640x200	16	80x25	8x8	1/2/4	A0000h
F	APA	MD	640x350	Mono	80x25	9x14	1/2	A0000h
10	APA	ECD	640x350	4	80x25	8x14	1/2	A0000h

```
Legend:
      MD = IBM Monochrome Display or equivalent
      CD = IBM Color Display or equivalent
     ECD = IBM Enhanced Color Display or equivalent
   1/2/4= 64K/128K/256K video memory installed
   2/4/8= 64K/128K/256K video memory installed
```

VGA video modes:

Mode	Type	Emul.	Resolution	Colors/ Palette	Text Scheme	Char Cell	Display Pages	Video Buffer
0,1	text	CGA	320x200	16/256K	40x25	8x8	8	B8000h
0,1	text	EGA	320x350	16/256K	40x25	8x14	8	B8000h
0,1	text	VGA	360x400	16/256K	40x25	9x16	8	B8000h
2,3	text	CGA	640x200	16/256K	80x25	8x8	8	B8000h
2,3	text	EGA	640x350	16/256K	80x25	8x14	8	B8000h
2,3!	text	VGA	720x400	16/256K	80x25	9x16	8	B8000h
4,5	APA	CGA	320x200	4/256K	40x25	8x8	1	B8000h
6	APA	CGA	640x200	2/256K	80x25	8x8	1	B8000h
7	text	MDA	720x350	MDA Mono	80x25	9x14	8	B0000h
7!	text	VGA	720x400	VGA Mono	80x25	9x16	8	B0000h
D	APA	EGA	320x200	16/256K	40x25	8x8	8	A0000h
E	APA	EGA	640x200	16/256K	80x25	8x8	4	A0000h
F	APA	EGA	640x350	Mono	80x25	8x14	2	A0000h
10	APA	EGA	640x350	16/256K	80x25	8x14	2	A0000h
11	APA	VGA	640x480	2/256K	80x30	8x16	1	A0000h
12	APA	VGA	640x480	16/256K	80x30	8x16	1	A0000h
13	APA	VGA	320x200	256/256K	40x25	8x8	1	A0000h

```
Legend:
     Mode 3! = power-on default mode when color monitor attached
     Mode 7! = power-on default mode when mono monitor attached
```

Figure 2

Typically, the character's dot pattern will utilize only a portion of the cell, leaving several pixels unused on all sides to enhance the character's readability.

Graphics Data Format

Data in the video buffer can also represent graphical images on the screen. However, graphics-oriented operating modes utilize video memory in a different manner than do text modes. Video memory is used to store color information for individual pixels on the screen, and a defined mapping scheme provides a correlation between any particular pixel on the screen and a location in video memory. To display graphical images on the screen, the computer's processor will work with the video adapter and simply read sequential video memory locations, translate the color information into a form suitable for sending to the display, and then send the information out to the screen for viewing.

The method at which color information is stored for each pixel is dependent on the number of colors possible for each pixel. For example, 4 bits are needed to store a unique number that represents one of 16 possible colors ($2^4 = 16$). In similar fashion, 8 bits are needed to uniquely identify one color from a selection of 256.

Video Modes

The PC world has several video adapters that are thought of as "standard". These standard adapters support numerous video modes, each having a mix of text and graphics modes, all with slightly different flavors in terms of resolution capabilities, color depth, character cell size, number of display pages, and starting address of the video memory buffer. Each adapter's operating modes are shown in Figure 2.

Adapter Hardware Components

All PC-compatible video adapters have specific functional components:

Video Buffer – Also known as "video memory", information to be displayed on the screen is stored here in a block of RAM that is mapped into the host processor's memory allocation map. The CPU memory area utilized for a video buffer is in the range A000h - BFFFFh, with the exact size and starting point dependent on adapter and operating video mode.

The size of this video buffer is also important as it determines how many colors can be displayed on the screen, and how many display pages can be stored. For example, standard VGA systems have 256Kb of video memory and VGA mode 12h allows for a 640x480 resolution with 16 colors. Storing color information for one pixel requires 4 bits, or 2 pixels per byte. Simple arithmetic tells us that 153,600 bytes of video memory are required to store one display page of information while in this mode. Displaying at 640x480 with

256 colors, a common "extended-VGA" mode (not IBM-compatible), would then require 300Kb because storing color information for one pixel requires 8 bits, twice that of standard VGA mode 12h, and beyond the 256K capability in standard VGA systems.

Example: Minimum memory required for 640H x 480V resolution:

16 color: 640 x 480 pixels x (1 byte/2 pixels) = 153,600 bytes = 150K

256 color: 640 x 480 pixels x (1 byte/1 pixel) = 307,200 bytes = 300K

Cathode Ray Tube Controller (CRTC) – The CRTC is the source of all horizontal and vertical timing signals required by the monitor.

Character Generator – A character generator takes the ASCII code of an alphanumeric character and translates it into a series of dots that can be sent out to the display screen. The generator contains dot patterns for every character that can be displayed. This character generator is in a non-accessible ROM for MDA, HGC, and CGA systems, and in video memory (downloaded from the ROM BIOS and/or from application programs) in EGA and VGA systems.

Attribute Decoder – Before "dots" orignating from the character generator are sent to the monitor for display, they can be modified with character attributes such as brightness, underlining, and blinking functions.

Video Signal Generator – There are three types of signals sent to the monitor, depending on the type of adapter and monitor being utilized. They are composite, digital, and analog. Each adapter has the capability to output signals as shown:

Adapter	**Output**
MDA	Digital
Hercules	Digital
CGA	Digital, Composite
EGA	Digital
VGA	Analog

ADAPTER TYPES

There are several types of video adapters used in IBM-compatible PCs, with varying degrees of resolutions, colors, and features. Next is an overview of the architecture and capability of each of the industry standard video adapters, plus a few non-standard ones as well.

Monochrome Display Adapter (MDA)

The Monochrome Display Adapter (MDA) was first introduced with the original IBM-PC. As its name implies, it was intended to work with the Monochrome Display, a single-color display found in offices everywhere. Its primary advantage is that it's a very low cost solution for displaying crisp, clear text. For most business and academic applications, this is adequate. However, it is limited in that it cannot display any graphics other than the crude "block" graphics that are part of the standard alphanumeric character set.

Address:					
B000:0000	char(0)	attribute(0)	...	char(4F)	attribute(4F)
B000:00A0	char(50)	attribute(50)	...		
	...	...	...		
B000:0F00	char(780)	attribute(780)	...	char(7CF)	attribute(7CF)

Figure 3

The heart of the MDA is a 6845 CRT controller and 4K bytes of dual-ported static RAM. The 6845 is a programmable device used primarily to handle the timing and control signals for the display system. The static RAM, more commonly called video RAM or video memory, is used as a buffer to store information on each character to be displayed on the screen. Since both the 6845 and the host CPU can access the video RAM (although not at the same time) the memory is considered to have two output ports, hence the term dual-port. This video memory contains character information that must be translated to a form more suitable for display on the screen.

From a programmer's view, the MDA looks like four I/O ports and a 4K byte block of memory starting at host address B000:0000. See Figure 3. With an 80 column x 25 line display, there are 2000 display locations that need to be described in memory. The video memory is set up in such a fashion that the first 2000 even-address memory locations contain the ASCII character code, while the odd-address memory locations contain the attributes of the characters. Therefore, 4000 of the 4096 bytes available are used to describe the contents of one display screen. Each display screen is known as one display page, so the MDA's 4K video memory supports only one display page.

One additional feature of the original MDA card was the inclusion of a parallel printer port. This option was added so that only one PC backplane slot was used to handle both video and printer functions.

Color Graphics Adapter (CGA)

IBM's first bit-mapped color adapter for the PC was their Color Graphics Adapter (CGA). It can operate in 40 or 80 column x 25 line text mode or in one of three bit-mapped graphics modes. The basic hardware architecture is very similar to the MDA's, with a 6845 CRT controller and 16K bytes of video memory.

Forty column text modes are available but are rarely used in PC operations anymore, and were never taken seriously by the computer industry, anyway. They were initially offered so that a television could be used in place of a computer monitor.

Functional operation of 80 column x 25 line text is essentially identical to that of the MDA. The methods used to store character information in memory and to translate it into a form that can be displayed are the same.

One important difference in text mode is the utilization of an 8 x 8 character cell. This smaller cell size reduces the readability of the text considerably. The characters within the cell typically use a 7 x 7 portion of the 8 x 8 cell leaving only one pixel for spacing. For this reason alone, many serious computer users will not use a CGA for text-based applications. The MDA is a far better choice.

The CGA's memory organization for text modes is similar to the MDA with two major differences: one is that the CGA has 16K of video memory versus 4K in the MDA, and its starting address is at B800:0000 rather than at B000:0000. The manner in which character information is stored is the same as for the MDA. Four thousand bytes (4000) are needed to store one page of text information. With 16K available, four display pages are possible.

The CGA also has three graphics modes from which to choose. They are, appropriately enough, low, medium, and high-resolution. The low resolution mode is 160 x 200 pixels. Oddly enough, CGA originator IBM does not support this mode. That is of little concern because no one really cares about this mode anyway, and few, if any CGA clone manufacturers support this mode either. It may allow for 16 colors, but the resolution is so low, that it is practically useless.

The medium resolution graphics mode is 320 x 200 pixels, with four simultaneous colors allowed. This mode is generally a compromise between resolution and color depth. It is probably the most popular CGA graphics mode, but it certainly is nothing to write home about.

The high resolution graphics mode is 640 x 200 pixels, but it only has 2-color capability. These two colors can be selected through the 6845's Color Select register.

The memory organization for CGA graphics modes is relatively simple and straightforward, similar to MDA, but certainly unique to CGA. Pixel information for all even-numbered display raster lines is stored in memory starting at location B800:0000 and information for odd-numbered raster lines starts at address

B800:2000H. Simply, the lower 8K is for even lines, the upper 8K for odd lines.

Video output from a CGA can be sent to one of three different types of displays. A 2-wire composite output connector allows the connection of a monochrome or color composite display that follows the NTSC standard. A similar composite-type output is available for a special demodulator for use with a television. These two methods are rarely used. The most useful, and popular, method is to connect a TTL monitor that is specially designed to operate with the CGA.

One of the most annoying problems associated with CGA has already been mentioned, that of text being difficult to read. Equally annoying is the obnoxious problem of "flicker" and "snow". The flicker is due essentially to the slow processing speed of the original PC and XT class of machines and shows up when the display screen is being scrolled while in high-resolution, 80-column text mode. The problem of "snow" is due to the CPU updating video memory during periods of time when the CRT controller is also accessing video memory for screen refresh. If memory is updated only during the display's vertical retrace periods, when the CRT controller is not accessing the memory, then snow could be eliminated. However, this retrace period is generally too short for a page of video memory to be completely updated when the CGA is installed in a slow PC or XT class of machine. One method at getting around this annoyance is to turn off the display's electron beam for the time necessary for the CPU to update the video memory for the entire display screen. This method eliminates the "snow" but it essentially robs Peter to pay Paul because it reintroduces our first problem: flicker.

From the progammer's view, the CGA looks like five I/O ports and a 16K byte block of memory starting at B800:0000.

Hercules Graphics Card (HGC)

Hercules Computer Technology, Inc. was one of the companies that took advantage of IBM's mistake in not having graphics capability on the MDA. If a customer wanted graphics, they had to purchase not only a CGA, but also a new display, because the CGA output was not compatible with the original Monochrome Display. On top of additional financial outlays to get graphics, the customer had to put up with some really terrible text quality in the CGA.

The originators of Hercules had a better idea that was so good at its time that it too, became a standard in a PC world dominated by IBM. The beauty of the Hercules Graphics Card (HGC) was its absolute emulation of the original IBM MDA card while operating in text mode, plus the added advantage of providing hi-resolution monochrome pixel graphics. The HGC was designed to take advantage of the user's existing monochrome display system and it even included a parallel printer port, just like the original MDA. The makers of the HGC also managed to get the support of Lotus Development Corporation in having the extremely popular Lotus 1-2-3 speadsheet package utilize HGC graphics.

Since the functionality of the HGC is so similar to the MDA, it should be no surprise that a 6845 CRT controller is utilized. But instead of putting a mere 4K or 16K of video memory on board, a full 64K was installed. This amount of memory allowed for 16 display pages while in text mode, and for two display pages while in graphics mode.

The 64K of memory was unusual in that it exceeded the memory space normally allocated for monochrome display systems. Instead, it was arranged as two banks of 32K, one situated at B000:0000, the other at B800:0000. This second bank conflicted with the memory space normally used by the CGA and later, the EGA and VGA. To prevent any problems in the event a CGA and a HGC were co-installed into the same PC, the HGC had a provision so that at boot-up it would allow access to only the first 32K memory bank at B000:0000. Using only this first half of memory is what Hercules calls its HALF graphics mode. When both banks are accessible, using a full 64K, it was considered to be in FULL graphics mode.

The HGC can be accessed using the same four I/O ports as the MDA, plus it has one extra I/O port peculiar to HGC, and through 64K of memory starting at B000:0000. The HGC has defined two additional bits in the CRT Control Port (03B8H) and there is also an additional Configuration port at 03BFH. A bit in the CRT Control Port determines whether the card operates in text or graphics mode, another bit selects the active display page while in graphics mode. The configuration port allows the user to enable or disable the second graphics page. Turning off the second page frees up the 32K memory block at B800:0000 and allows the user to co-install an EGA or CGA board.

Enhanced Graphics Adapter (EGA)

In 1984, IBM introduced its Enhanced Graphics Adapter (EGA). This video solution was widely welcomed as it provided a superior solution for MDA and CGA users alike. The primary intent of the EGA was to provide higher resolution and greater color depth, emulate the MDA and CGA, and to eliminate the need for two standards: MDA for text and CGA for color and graphics. Using the EGA allowed a user to get by with one video adapter for all existing text and graphics modes, plus it added a few higher resolution modes, and it allowed the user to continue using existing display monitors. Note that the EGA can emulate the MDA and CGA. The MDA and CGA use 6845 CRT controllers while the EGA does not. The problem this poses is that software that directly manipulates the MDA and CGA register set stands a very good chance of NOT running on the EGA.

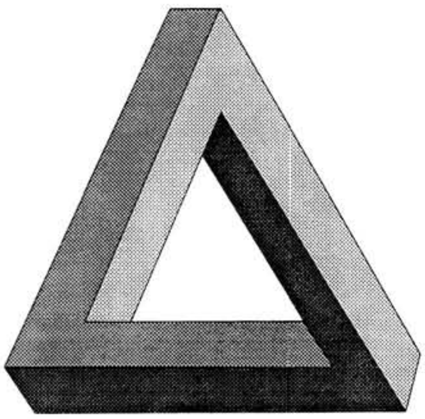
With the increase in display resolution capability to 640 x 350 pixels, the existing text modes were enhanced. An 8x14 character block size was chosen, with the actual character typically using a 7x9 matrix within the block. This brought about a drastic improvement in text quality when the EGA emulates any of the CGA text modes (0-3). Emulation of the MDA in mode 7 did not change, however. It still utilized a 720 x 350 pixel resolution and a 9x14 character block.

EGA graphics were also greatly enhanced, both in functionality and in complexity. The increased resolution capability allowed for four new video modes, three color and one monochrome. The color modes take advantage of the increased resolution and color capabilities inherent to the EGA while the monochrome mode was IBM's attempt at establishing a monochrome graphics standard. Note that in this attempt at establishing their own monochrome graphics standard, IBM has virtually ignored the Hercules standard. Most manufacturers of EGA clones however do not ignore the fact that HGC compatibility is an important feature and have likewise included HGC graphics capability in their board products.

IBM also designed the EGA so that it would play on any standard IBM monitor, whether it be the old, original Monochrome Display (MD), the Color Display (CD), or the newer Enhanced Color Display (ECD). This feature precluded the necessity for users to go out and buy a new display. The Monochrome Display could be used to display MDA compatible text mode 7 or the new, high-resolution monochrome graphics mode 0FH. The Color Display can be used to display any of the existing CGA-level modes 0-6. The Enhanced Color Display is necessary to utilize any of the enhanced color modes introduced by the EGA (modes 0DH, 0EH, and 10H).

One of the unique features of the EGA architecture is its implementation of video memory within the confines of the PC memory allocation map. The EGA was designed to accomodate anywhere from 64K to 256K of video memory. And with the advent of higher display resolutions and greater color depth, more physical memory was needed than was available in the PC's memory allocation. IBM came up with a novel method of storing video information in several "banks" of physical memory and then bank-switched a section of this physical memory into an available PC memory space as needed. The EGA is able to "see" four banks, or planes, of

Continued on Page 35



ASSEMBLY LANGUAGE

Part 10 Disk I/O

Pat Swayne
ZUG Software Engineer

This is the tenth installment in my hit-or-miss series on assembly language. Those of you who have been following the series should be familiar with the 8086 instruction set, and you should be able to write simple programs that can read the keyboard and/or output to the screen. In this article, I will discuss how to read and write disk files.

Disk I/O Methods

In MS-DOS, there are three different ways provided to read and write disk files. These are the FCB method, the file handle method, and the I/O redirection method. The FCB (File Control Block) method is a carry-over from the old CP/M 8-bit operating system, and it was the only method available in version 1 of MS-DOS. However, since the introduction of MS-DOS 2 and more versatile disk I/O methods, the FCB method has become obsolete, and we will not discuss it here. We may discuss file control blocks in connection with other tasks in the future, however, because there are a few things that are best done with them.

The file handle method is the one most commonly used these days. This method allows files to be specified using complete paths, whereas the FCB method only works with file names.

Disk Operations Using I/O Redirection

The I/O redirection method is the easiest one to work with, so it is the first method I will discuss. This method uses the ability of MS-DOS to redirect the standard input and output devices (the keyboard and screen) to other "devices", or files. In other words, the MS-DOS functions that you have already learned for inputting from the keyboard and writing to the screen can be used to read and write files.

To illustrate how easy it is to read and write disk files using I/O redirection, suppose MS-DOS had no COPY command and you had to write your own. Figure 1 shows a program that can do the job.

```
CODE     SEGMENT
         ASSUME   CS:CODE,DS:CODE,ES:CODE,SS:CODE
         ORG      100H

START:   MOV      AH,0BH
         INT      21H              ;CHECK DEVICE STATUS
         OR       AL,AL            ;ANY CHARACTER?
         JZ       DONE             ;IF NOT, EXIT
         MOV      AH,8
         INT      21H              ;GET CHARACTER, NO ECHO
         MOV      DL,AL
         MOV      AH,2
         INT      21H              ;OUTPUT CHARACTER
         JMP      START
DONE:    INT      20H              ;EXIT

CODE     ENDS
         END      START
```

Figure 1. XFER, a simple file copy program.

Call the assembly language file XFER.ASM, and assemble it to XFER.COM. To use it to copy files, enter

```
XFER <source >destination
```

The source and destination can be any valid path, as long as the file name is included in each path, and you include the < and > symbols. This program does not do "wild cards", and it cannot be used to copy from the CON device (the keyboard) to a file because of the status check which is used to check for the end of a file. If you experiment with XFER, you will find that it is slower than the COPY command. In fact, it is a rather inefficient way to copy files, not only because of the overhead involved in I/O redirection, but because it only copies one character at a time. If your system has a disk cache, that will speed things up. If you are still using MS-DOS version 2, be prepared to wait a L-O-N-G time for the program to work. Another problem with XFER is that it does not preserve the date/time stamp of the original file. The copy will be stamped with the current date and time. Actually, a file copy program is a comparatively difficult program to write, if it is to be as versatile as the DOS COPY command.

Another somewhat serious problem with XFER is that if the source file contains a byte with the number 3 in it (the value 3, not the ASCII character 3), XFER will stop copying and exit when it reaches that byte. That is because 3 is the ASCII value of Control-C, and the I/O functions used in XFER are set up to exit to MS-DOS when a Control-C is detected. This problem is fixed if function 3Fh is used for input. The companion function 40h can be used for output. Figure 2 shows a version of XFER that uses these functions.

This version of XFER not only fixes the Control-C problem, but it can also be used to copy from the CON device to a file.

You may be thinking that the I/O redirection method of working with files is not good for much, but actually it does serve well for one type of program – the filter. A filter is a program that reads text or other data, alters it somehow, and writes it back out. For example, there are filters that convert tabs to spaces and spaces to tabs. In fact the first version of XFER actually is a filter that converts tabs to spaces. If you entered the source code for it using tab characters instead of spaces between the

```
CODE    SEGMENT
        ASSUME  CS:CODE,DS:CODE,ES:CODE,SS:CODE
        ORG     100H

START:  MOV     BX,0               ;STD INPUT HANDLE
        MOV     DX,OFFSET BUFFER   ;PUT CHARACTER HERE
        MOV     CX,1               ;READ ONE CHARACTER
        MOV     AH,3FH
        INT     21H                ;READ A CHARACTER
        OR      AX,AX              ;ANYTHING READ
        JZ      DONE               ;IF NOT, EXIT
        MOV     CX,1
        MOV     AH,40H
        MOV     BX,1               ;STD OUTPUT HANDLE
        INT     21H                ;WRITE CHARACTER
        JMP     START
DONE:   INT     20H                ;EXIT
BUFFER  LABEL   BYTE               ;BUFFER CHARACTERS HERE

CODE    ENDS
        END     START
```

Figure 2. A version of XFER that uses functions 3F and 40.

```
CODE    SEGMENT
        ASSUME  CS:CODE,DS:CODE,ES:CODE,SS:CODE
        ORG     100H

START:  MOV     BX,0               ;STD INPUT HANDLE
        MOV     DX,OFFSET BUFFER   ;PUT CHARACTER HERE
        MOV     CX,1               ;READ ONE CHARACTER
        MOV     AH,3FH
        INT     21H                ;READ A CHARACTER
        OR      AX,AX              ;ANYTHING READ
        JZ      DONE               ;IF NOT, EXIT
        AND     BYTE PTR BUFFER,7FH ;STRIP HIGH BIT
        MOV     CX,1
        MOV     AH,40H
        MOV     BX,1               ;STD OUTPUT HANDLE
        INT     21H                ;WRITE CHARACTER
        JMP     START
DONE:   INT     20H                ;EXIT
BUFFER  LABEL   BYTE               ;BUFFER CHARACTERS HERE

CODE    ENDS
        END     START
```

Figure 3. STRIP, a simple filter program.

columns, try copying it with the first XFER and then examine the copy with a program that can distinguish between tab characters and spaces (some word processors can do that, or you can use the D (dump) command of DEBUG).

By adding one line of code to the second XFER program, you can change it into a filter that strips the high bit from characters in the input file. The modified program is shown in Figure 3.

Call this modified version of the file STRIP.ASM, and assemble it to STRIP.COM. Make sure that the output name you specify is not the same as the input name, or on a different drive/directory, when you run this program.

One thing you can do to greatly improve the efficiency of a program like STRIP is to buffer the input and output. In other words, make it read and write more than one byte at a time. For a filter like STRIP, which outputs the same number of bytes that it inputs, buffering is fairly easy to do. Figure 4 shows a version of STRIP that buffers input and output.

Enter this program as STRIP2.ASM, and assemble it to STRIP2.COM. STRIP2 assumes that you have at least 64k of free memory in your computer. If you do not have that much, the program will probably "crash" if you try to strip a large file. To compare the speed of the two STRIP versions, try running a file about 8k in size through each version.

It is relatively easy to add buffering to a program that does not add or subtract bytes from a file as it processes it, but a filter that does add or subtract bytes, such as a tabs to spaces filter, is more difficult to program with buffering. With this type of program, you need two buffers – one to hold the input, and one to hold the output. I find that using a large buffer for input, and a smaller buffer for output (1k for example) works fine for these programs, as does having two moderately sized buffers (16k for example). If you would like to study the source code for some two buffer filter programs, get my "Yet Another Utilities Disk", ZUG p/n 885-6015.

The Handle Method

The handle method of file I/O is like the I/O redirection method when functions 3F and 40 are used, except that additional work must be done. The file must be "opened" before it can be read or written, and "closed" after the operation. To illustrate the handle I/O method, I have written a "real" little program called OneCopy, which is listed following this article. Type in the listing as OC.ASM, and assemble it to OC.COM. OC is used to copy files from one floppy disk to another using only one floppy drive. To use OC, enter

```
OC filename
```

where filename is the name of the file or files you want to copy. You can use wild card characters (* and ?) to specify more than one file to copy. You can also specify a path. OC will prompt you with "Insert source, press Enter...". Insert the disk containing the file(s) to copy and press Enter. OC will read the first file to copy into memory, and prompt with "Insert destination, press Enter...". Remove the disk containing the file, and replace it with the disk you want to copy to and press Enter. OC will copy the file to the disk, and prompt for the source disk again. If there are more files to copy, it will continue this process until all files are copied. OC only uses 64k of memory for itself and the buffer space for copying files, so if you are copying a large file, it may prompt you to insert the source and destination several times for a single file. OC lists the name of each file on the screen as it copies it.

If OC finds the same path on the destination disk that you specified on the source disk, it will put the copied file(s) there. If it cannot find the path, it will put the file(s) in the root directory of the destination disk. If a file with the same name as one your are copying already exists on the destination disk, OC will prompt with "File exists, delete?". If you type Y, the file will be deleted and replaced with the new one. If you type N, OC will go on to the next file (if any), or quit. If OC encounters a problem trying to write out a file, it will print an error message and go on to the next file, or quit.

Here is an explanation of how OC works. Before the start of the program are some labels defined using ORG statements. The first one, ZERO is defined because of a quirk in the MS-DOS assembler

```
CODE    SEGMENT
        ASSUME  CS:CODE,DS:CODE,ES:CODE,SS:CODE
        ORG     0
ZERO    LABEL   NEAR
        ORG     100H

START:  MOV     SP,OFFSET START ;PUT STACK HERE
CPYLP:  MOV     BX,0            ;STD INPUT HANDLE
        MOV     DX,OFFSET BUFFER ;PUT DATA HERE
        MOV     CX,ZERO-BUFFER  ;READ BUFFER FULL
        MOV     AH,3FH
        INT     21H             ;READ A CHARACTER
        OR      AX,AX           ;ANYTHING READ
        JZ      DONE            ;IF NOT, EXIT
        MOV     SI,OFFSET BUFFER ;POINT TO BUFFER
        MOV     CX,AX           ;COUNT TO CX
STRLP:  AND     BYTE PTR [SI],7FH ;STRIP HIGH BIT
        INC     SI
        LOOP    STRLP
        MOV     CX,AX           ;GET COUNT AGAIN
        MOV     AH,40H
        MOV     BX,1            ;STD OUTPUT HANDLE
        INT     21H             ;WRITE DATA
        JMP     CPYLP           ;COPY REST OF FILE
DONE:   INT     20H             ;EXIT
        EVEN
BUFFER  LABEL   BYTE            ;BUFFER DATA HERE

CODE    ENDS
        END     START
```

Figure 4. A version of STRIP that uses I/O buffering.

and its clones. The second label, ARG, is the location of any arguments that you type after the program name when you run the program. When you run a program in MS-DOS, those "command line arguments" are placed at location 80h in the program segment prefix (PSP), a 256-byte block of memory that MS-DOS creates for each program it runs. In a .COM file, the PSP is always the first 256 bytes before the beginning of the program and so the command line arguments are always at ORG 80H.

The location 80h in the PSP is also an area called the "default DTA". DTA stands for Disk Transfer Address, and if we were using the FCB method of file I/O, the DTA would be the area where data read from a file would be placed, or where data written to a file would be expected. It also happens to be the place where data is placed when you use the MS-DOS function for searching for file names in a directory. The actual file name is stored at 1Eh bytes offset from the start of the DTA, so we have a label SNAME defined at that location.

Notice in the first line of the program that the stack pointer is set to the label START. Normally in a .COM file, the stack pointer is at the top of the 64k segment of memory that MS-DOS gives to the .COM file. OC needs to use all of the space above itself for data, so that is why the stack is placed at the label START, where it will grow downwards.

The next few lines of the program check the command line for an argument. The first byte of the command line argument stored at ARG is a count of the characters in the argument, with the actual argument following. OC checks this count, and if it is zero, it displays a message explaining how to use the program. If the count is not zero, OC skips over any spaces at the beginning of the argument and copies the remainder to an area called BA.PATH. If you look through the rest of the program, you will not find a label called BA.PATH, but you will find a label called BA, and below that another label, PATH. Before the label PATH is an assembler directive that may be new to you, STRUC. This is the directive for setting up a data structure, and it allows you, among other things, to reserve memory space in a data area without actually reserving space in your program. If I had simply had

```
PATH    DB      80 DUP (?)
BUFFER  DB      ?
```

in the program without the STRUC directive, it would have created a .COM file 81 bytes longer than it currently does. The assembler would have placed the bytes defined in the DB statements in the program, even though they are just "garbage" bytes that will be overwritten when the program is run. You can think of the period in the compound label BA.PATH as a plus sign. The value of the label BA, which is determined by its place in the program, is added to the value of the label PATH, which is determined by its place in the structure, not the program. Actually, the label PATH has a value of zero, so we could have used BA by itself, but the value of the other label in the structure, BUFFER, has a value of 80. So when you see BA.BUFFER later in the program, you will know that it has the value, or address, of BA plus 80.

After the argument is copied to BA.PATH, it is scanned for the characters ":" and "\", so the program can determine if a drive name and/or path is specified before the actual file(s) to copy. OC needs this information so it can copy files into the root directory of the destination if the source directories do not exist. When the argument is copied to BA.PATH, it is terminated with a zero. A string of text that is terminated with a zero is called ASCIZ text, and many of the newer MS-DOS functions (since 2.0) require an argument in ASCIZ text.

OC prepares to enter the main copy loop by prompting for the source disk and searching for the file or the first file (if wild cards were specified) to copy. In MS-DOS, there are two sets of directory search functions – an old set that uses FCB's, and a newer set that uses ASCIZ strings. I will only use the newer set here. In each set there are two functions, a search first function and a search next function. If the search string contains wild cards, the search first function will find the first matching directory entry and store information about it at the DTA location, as I pointed out previously. The search next function will find the next and subsequent matching entries each time it is used. So at the label MAINLP, the search first function is used the first time through the loop, and the search next function is used every other time. The search next function is loaded into the AH register at the label DONE, just before the jump back to MAINLP.

If OC does not find a matching entry the first time through the loop, it indicates "File not found," and exits. If it does not find an entry any other time through, it just exits. A memory location called FTFLG (First Time Flag) is used to indicate when the first time through the directory search is done.

If a matching directory entry is found, the file name is copied to two places. It is copied to the end of the complete path that the user specified, and it is also copied to another area that just includes the user-specified drive, if any. Then OC displays the file that is about to be copied and enters the file copy loop.

In the file copy loop, OC opens the file by using the supplied path with the specific file name copied to the end of it as the required ASCIZ string. The open function returns a handle number in AX, which is copied to the BX register for use in the "read" and other functions. Then OC uses a special function sometimes called LSEEK that positions a pointer which determines where MS-DOS will start reading from or writing to a file. It uses the value in a location called PASSCNT multiplied by the size of the file buffer to locate this pointer. Of course, the first time through, PASSCNT will contain a zero (it was zeroed one line below the label GOTFIL), so the pointer will be at the beginning of the file. Next, OC reads the date and time from the file using a special function for that and stores the information for placement in the destination file, so that it will have the same date and time as the source. Then OC reads as much of the file as will fit into its buffer.

After OC reads in a "chunk" of the file, it closes it. This step would not be necessary in a normal file copy utility, but this program has to write the output file onto another disk in the same drive as the source. All kinds of damage could be done if it did not close files after each read and write operation.

When OC has finished reading from the source and closed the file, it prompts the user to change disks. Then it attempts to open a file for writing using the user's path with the specific name at the end. If this is not successful, it tries again using the string containing just the file name and the drive, if any. In this way the copy will be successful whether a duplicate path exists on the destination drive or not. You will also notice that OC uses two different functions in its attempts to open the destination file. The "open" function will be successful only if the file already exists on the destination disk. Otherwise, the "create" function must be used. If a file is larger than OC's buffer, it will have to make more than one "pass" to copy it. It will use the create function on the first pass, and the open function on subsequent passes. The pass counter at PASSCNT is used to keep track of the copy passes, and to position the file pointer for each pass.

Rather than determining how big a file is before it copies it, OC just loops back through the copy loop for at least a second time on every file. If the second attempt to read from the file returns a byte count of zero, then OC knows that it has completed that file and goes back to the main loop, where the search next function will determine if there are any more files to copy.

If there are any errors in attempting to open or write to the destination file, OC will quit the attempt on that file and display an error message. This is rather simple error handling that could be improved on a bit.

After OC writes each piece of the file it is copying to the destination, it sets the date and time using the information obtained from the source file. Notice that the same function is used to read and set the date and time. The value in the AL register determines what the function will do.

Following the main part of the program are a few subroutines for prompting for disk insertions, and showing the name of the file OC is currently working on. Then there is the data area, which contains the text messages used in the program, and the flags, counters, and storage areas.

Although this program works, and is somewhat useful, there are some significant improvements that could be made to it. The copy buffer could be made to use all available memory, not just a 64k segment. Reading into and writing from this enlarged buffer would involve manipulation of the segment registers. Another improvement would be to have the program copy all the files that will fit into the buffer before it has you swap disks. If you are copying a hundred 1k files, swapping disks all those times could get old fast. The program would have to keep track of the beginning and end of each file in memory, and it would have to store the names, dates and times of all of the files. If you would like to try these changes, send me what you come up with, and if yours is the best version I will print it in REMark, give you credit, and perhaps something else for your effort.

This concludes this installment on assembly language. I'm not sure what I will cover in the next one, so if you have ideas, let me know.

```
;       ONECOPY PROGRAM
;       BY P. SWAYNE, ZUG SOFTWARE ENGINEER

CODE    SEGMENT
        ASSUME  CS:CODE,DS:CODE,ES:CODE,SS:CODE
        ORG     0
ZERO    LABEL   NEAR
        ORG     80H
ARG     LABEL   BYTE            ;COMMAND LINE ARGUMENT
        ORG     80H+1EH
SNAME   LABEL   BYTE            ;SEARCH NAME
        ORG     100H

;       PROCESS COMMAND LINE

START:  MOV     SP,OFFSET START ;PUT STACK HERE
        CLD                     ;CLEAR DIR. FLAG
        MOV     SI,OFFSET ARG   ;POINT TO USER ARG
        LODSB                   ;GET CHARACTER COUNT
        OR      AL,AL           ;ANY ARGUMENT?
        JZ      EXPL            ;IF NOT, EXPLAIN PGM.
        MOV     CL,AL
        XOR     CH,CH           ;COUNT TO CX
SOS:    CMP     BYTE PTR [SI],' ' ;SPACE?
        JNZ     NOTSP           ;NO
        INC     SI
        LOOP    SOS             ;ELSE, SKIP SPACES
EXPL:   MOV     DX,OFFSET EXPMSG
        MOV     AH,9
        INT     21H             ;EXPLAIN PROGRAM
        INT     20H             ;AND EXIT
NOTSP:  MOV     BX,CX           ;COUNT TO BX
        MOV     DI,OFFSET BA.PATH
        PUSH    DI
        REP     MOVSB           ;COPY ARGUMENT TO HERE
        XOR     AL,AL
        STOSB                   ;PUT ZERO AT END
        POP     DI
        MOV     SI,DI           ;SAVE POINTER IN SI
        ADD     DI,BX           ;POINT TO END OF PATH
        MOV     AL,':'          ;SEARCH FOR THIS
        MOV     CX,BX           ;GET COUNT
        STD                     ;SEARCH BACKWARDS
        REPNZ   SCASB           ;LOOK FOR ":"
        CLD                     ;FIX DIR. FLAG
        JNZ     NODRV           ;NO DRIVE
        MOV     AL,[DI]
        MOV     DNAME,AL        ;ELSE, FIX DEST. NAME
        MOV     BYTE PTR DNAME+1,':'
        INC     DI
        INC     DI
        MOV     FNAME,DI        ;MARK DESTINATION PATH
```

```
NODRV:  MOV     DI,SI               ;FIX POINTER
        ADD     DI,BX               ;PNT. TO END OF PATH
        MOV     AL,'\'              ;SEARCH FOR THIS
        MOV     CX,BX               ;GET COUNT
        STD                         ;SEARCH BACKWARDS
        REPNZ   SCASB               ;LOOK FOR "\"
        CLD                         ;FIX DIR. FLAG
        JNZ     NOPTH               ;NO PATH
        INC     DI                  ;POINT PAST \ OR :
        INC     DI
        MOV     FNAME,DI            ;SET UP POINTER

;       LOOK FOR FILES TO COPY

NOPTH:  CALL    INSSRC              ;ASK FOR SOURCE DISK
        MOV     AH,4EH              ;SEARCH FOR FIRST
        XOR     CX,CX               ;NO ATTRIBUTES
        MOV     DX,OFFSET BA.PATH ;POINT TO PATH

;       MAIN COPY LOOP

MAINLP: INT     21H                 ;SEARCH FOR FILE
        JNC     GOTFIL              ;WE HAVE IT
        CMP     BYTE PTR FTFLG,0 ;FIRST TIME DONE?
        JNZ     NOTFT               ;NO
        MOV     DX,OFFSET NOFIL
        MOV     AH,9
        INT     21H                 ;ELSE, SAY "NO FILE"
NOTFT:  INT     20H

;       FOUND A FILE, COPY IT

GOTFIL: MOV     BYTE PTR FTFLG,1    ;FIRST TIME DONE
        MOV     BYTE PTR PASSCNT,0 ;CLEAR PASS COUNT
        MOV     SI,OFFSET SNAME ;POINT TO SRCH NAME
        PUSH    SI
        MOV     DI,FNAME            ;PUT IT HERE
        MOV     CX,13
        REP     MOVSB               ;COPY NAME INTO PATH
        POP     SI
        MOV     DI,OFFSET DNAME ;COPY HERE, TOO
        CMP     BYTE PTR DNAME+1,':' ;GOT DRIVE?
        JNZ     CPYNAM              ;NO
        ADD     DI,2                ;ELSE, SKIP OVER IT
CPYNAM: MOV     CX,13
        REP     MOVSB
        MOV     DX,OFFSET COPYING
        MOV     AH,9
        INT     21H                 ;SAY "COPYING..."
        CALL    SHOPTH              ;SHOW FILE
        JMP     CPYLP1              ;COPY THE FILE

;       FILE COPY LOOP

CPYLP:  CALL    INSSRC              ;ASK FOR SOURCE DISK
CPYLP1: MOV     DX,OFFSET BA.PATH ;POINT TO PATH
        MOV     AX,3D00H
        INT     21H                 ;OPEN SOURCE FILE
        MOV     BX,AX               ;HANDLE TO BX
        MOV     AL,PASSCNT          ;GET PASS COUNT
        XOR     AH,AH               ;IN AX
        MOV     DX,ZERO-BA.BUFFER
        MUL     DX                  ;CALCULATE FILE POS.
        MOV     CX,DX
        MOV     DX,AX               ;CX:DX = POSITION
        MOV     AX,4200H
        INT     21H                 ;POSITION FILE PNTR
        MOV     AX,5700H
        INT     21H                 ;GET SRCE DATE/TIME
        MOV     TIME,CX             ;SAVE THEM
```

```
        MOV     DATE,DX
        MOV     DX,OFFSET BA.BUFFER ;PUT DATA HERE
        MOV     CX,ZERO-BA.BUFFER ;READ BUFFER FULL
        MOV     AH,3FH
        INT     21H                 ;READ FROM THE FILE
        PUSH    AX
        MOV     AH,3EH
        INT     21H                 ;CLOSE FILE
        POP     AX
        OR      AX,AX               ;ANYTHING READ
        JZ      DONEJ               ;IF NOT, EXIT
        MOV     BYTES,AX            ;SAVE BYTES READ
        CALL    INSDST              ;ASK FOR DESTINATION
        MOV     DX,OFFSET BA.PATH ;POINT TO PATH
REOPEN: MOV     AX,3D01H
        INT     21H                 ;OPEN DESTINATION
        MOV     BX,AX               ;HANDLE TO BX
        JNC     OPNDOK              ;OPENED OK
        CMP     AX,3                ;BAD PATH?
        JNZ     TRYCR               ;NO, TRY CREATING
        MOV     DX,OFFSET DNAME ;GET NAME W/O PATH
        JMP     REOPEN              ;AND TRY AGIAN
TRYCR:  MOV     AH,3CH
        XOR     CX,CX
        INT     21H                 ;CREATE NEW FILE
        MOV     BX,AX               ;HANDLE TO BX
        JNC     OK2WRT
        JMP     WRTERR              ;WRITE ERROR
OPNDOK: CMP     BYTE PTR PASSCNT,0 ;FIRST PASS?
        JNZ     OK2WRT              ;NO, OK TO WRITE
        MOV     DX,OFFSET EXISTS
        MOV     AH,9
        INT     21H                 ;ESLE, SAY "EXISTS"
        MOV     AH,1
        INT     21H                 ;INPUT A KEY
        AND     AL,5FH              ;CAPITALIZE
        CMP     AL,'Y'              ;IS IT "Y"?
        JZ      OK2WRT              ;YES
        MOV     AH,3EH
        INT     21H                 ;ELSE, CLOSE FILE
DONEJ:  JMP     DONE                ;TRY NEXT FILE
OK2WRT: MOV     DX,OFFSET CRLF
        MOV     AH,9
        INT     21H                 ;PRINT CRLF
        MOV     AL,PASSCNT          ;GET PASS COUNT
        XOR     AH,AH               ;IN AX
        MOV     DX,ZERO-BA.BUFFER
        MUL     DX                  ;CALCULATE FILE POS.
        MOV     CX,DX
        MOV     DX,AX               ;CX:DX = POSITION
        MOV     AX,4200H
        INT     21H                 ;POSITION FILE PNTR
        MOV     CX,BYTES            ;GET BYTE COUNT
        MOV     DX,OFFSET BA.BUFFER ;DATE IS HERE
        MOV     AH,40H
        INT     21H                 ;WRITE DATA
        JNC     WRTOK
        JMP     WRTERR              ;WRITE ERROR
WRTOK:  MOV     CX,TIME
        MOV     DX,DATE             ;GET TIME AND DATE
        MOV     AX,5701H
        INT     21H                 ;SET THEM ON OUTPUT
        MOV     AH,3EH
        INT     21H                 ;CLOSE FILE
        JNC     CLOK
        JMP     WRTERR              ;WRITE ERROR
CLOK:   INC     PASSCNT             ;INCR PASS COUNT
        JMP     CPYLP               ;COPY REST OF FILE
DONE:   MOV     AH,4FH              ;GET SRCH NEXT FUNC.
        JMP     MAINLP              ;LOOK FOR MORE FILES
```

```
WRTERR: MOV     DX,OFFSET WERMSG
        MOV     AH,9
        INT     21H             ;SAY "WRITE ERROR"
        CALL    SHOPTH
        MOV     DX,OFFSET CRLF
        MOV     AH,9
        INT     21H             ;PRINT CRLF
        JMP     DONE            ;DO NEXT FILE

;       SUBROUTINES
;       PROMPT FOR SOURCE

INSSRC: MOV     DX,OFFSET INSSMSG
INSCOM: MOV     AH,9
        INT     21H             ;ASK FOR SOURCE
WFCR:   MOV     AH,8
        INT     21H             ;GET A KEY
        CMP     AL,13           ;CR?
        JNZ     WFCR
        RET

;       PROMPT FOR DESTINATION

INSDST: MOV     DX,OFFSET INSDMSG
        JMP     INSCOM          ;ASK FOR DESTINATION

;       SHOW PATH

SHOPTH: MOV     SI,OFFSET BA.PATH ;POINT TO PATH
SHOPTH1:LODSB                   ;GET A CHARACTER
        OR      AL,AL           ;END OF PATH?
        JZ      PTHDN
        MOV     DL,AL
        MOV     AH,2
        INT     21H             ;SHOW CHARACTER
        JMP     SHOPTH1
```

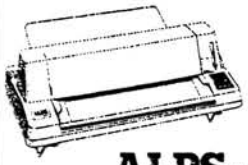
```
PTHDN:  RET

;       DATA AREA

EXPMSG  DB      13,10,'OneCopy version 1.0',13,10
        DB      'To use this program, enter'
        DB      13,10,10
        DB      '  OC <file>',13,10,10
        DB      'where <file> is the name of the'
        DB      ' file to copy.'
CRLF    DB      13,10,'$'
COPYING DB      13,10,'Copying $'
NOFIL   DB      13,10,'File not found.',13,10,'$'
EXISTS  DB      13,10,'File exists, delete? $'
INSSMSG DB      13,10,'Insert source, press'
        DB      ' Enter...$'
INSDMSG DB      13,10,'Insert destination, press'
        DB      ' Enter...$'
WERMSG  DB      13,10,'Error writing file $'
DNAME   DB      'A;FILENAME.EXE',0
FNAME   DW      OFFSET BA.PATH  ;FILE NAME POINTER
FTFLG   DB      0               ;FIRST TIME FLAG
BYTES   DW      0               ;BYTES READ
PASSCNT DB      0               ;PASS COUNT
TIME    DW      0               ;SAVED FILE TIME
DATE    DW      0               ;SAVED FILE DATE
        EVEN
BA:
BUFF    STRUC
PATH    DB      80 DUP (?)
BUFFER  DB      ?               ;BUFFER DATA HERE
BUFF    ENDS

CODE    ENDS
        END     START
```

W S Electronics

(513) 376-4348 ** Since 1975 **** (513) 427-0287**

1106 State Route 380, Xenia, Ohio 45385

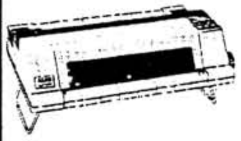
YOU AND YOUR BIG IDEAS.



* 300 cps draft speed
* Wide carriage
* 24 pin printing
* Front panel controls

ALPS *Allegro 500XT™*

THE ALPS ASP1600 PRINTER. COSTS VERY LITTLE, JAMS NOT AT ALL.



• Auto tear bar prevents paper waste. With the flick of a button, your fanfold paper advances to the perforation, then automatically returns to top-of-form position.

• Rugged 9-pin head delivers crisp output at 192 cps in draft mode, 38 cps in letter quality.

• Compact 9-lb. body makes for easy portability.

• Printer stand is built-in.

• Prints labels easily.

• Full Epson FX-85 companbility.

ALPS AMERICA
Built by popular demand.

SPECIAL

Z-315	EMS Kit for Z-159	$ 10.00
Z-417	H D Controller Z-248	$100.00
Z-317	H D Controller Z-150	$ 75.00
ZCM-1400-1	Swivel Base for 1490	$ 5.00
CB-4364-39	Diagnostics for 184	$ 5.00
ENABLE 2.0	Word Processing	$ 99.00

Quantities limited to stock on hand

ATTENTION:

Federal Government Offices
We stock ALL ALPS Printer Models and we stock ALPS PARTS and RIBBONS for all ALPS models including your P2000's and ASP-1000's

****GOVERNMENT DISCOUNTS OFFERED****

We are looking for good dealers.
ALPS Authorized Distributor and Service Center

Enable Revisited

Part Six
DataBase - A Beginning

George Elwood
1670 N. Laddie Court
Beavercreek, OH

In the last article, I briefly covered the advanced graphing capability of Enable. Advanced graphing is available for the spreadsheet while the basic graphing capability is available for both the spreadsheet and database. This article will cover the Enable database. This is one of the strongest areas of Enable and I have used it extensively.

The Enable database provides you with the ability to create database applications with a minimum amount of effort. I created a menu-driven database for a military hospital with special functions to track both patient's and medical technician's time in a pressure chamber. I wrote the output forms to match the paper forms in use. I created this whole application which consisted of four databases, three menus with the appropriate options, and forms in less than two hours.

I also wrote an application for a tool and die company, which they use to manage their entire operation, in about 200 hours. This again is completely menu driven with some of the users of the system having minimum computer knowledge. This application handles all operations from quotes to orders, job cards to payroll, and invoicing to packing lists.

In this article, I will move through the capabilities of the Enable database and show you some of the powerful tools available. I will use some parts of the tool and die application to illustrate these functions.

Before starting to design a database for any purpose, you must first think through your requirement. The time you spend working through this step will save you time later. It is a lot easier to add another field to a database before records have been added. Think of the amount of time and effort it would take to add another field and then add the data to a 1000-record database.

When I teach database, the design step is a foot stomper and then some. I stress this step to all of my students. The way I design database applications is to decide what outputs I am going to need. I then start designing the databases. Because Enable is easy to use, I seldom use only one database for an application. Enable's multiple database functions make this task easy to accomplish. I lay out the fields I will need and the sizes I feel will be sufficient to meet the requirements on paper and wait sometime to check for changes. I then review the database and I almost always make changes.

Another way to accomplish this review is to work with several people. I host the Dayton Enable Users Group. One of the members had a requirement for a database to control a warehouse function. It was a small operation but had several unique Air Force requirements. Working in a brainstorming session, the entire database and field requirements were defined in less than an hour. Many people provided helpful hints and suggestions to speed up this process. Again, the application consisted of several individual databases linked together.

Enable OA's database has been improved over earlier versions. The number of records in a database is now limited only to disk space. I am aware of one Enable database with several hundred thousand records, which resides on two 650Meg Hard Drives.

The database function now supports MEMO fields. Unlike dBase, the MEMO field is limited only by disk space. The information inserted in the memo field is maintained in an ASCII format so it is simpler to access. Enable's database information file is fully compatible with dBase III. It is also possible to make the database information dBase II compatible if desired. This compatibility means that the database could have been created with one of the dBase products and then used with Enable. You could work with the database with Enable and still use the basic file with the dBase product.

Unlike dBase and some of the other database programs on the market, Enable's database is made up of two parts. The first part, the .$BF file, holds the field parameters, and the .BDF file holds the actual records. This is the part that is com-

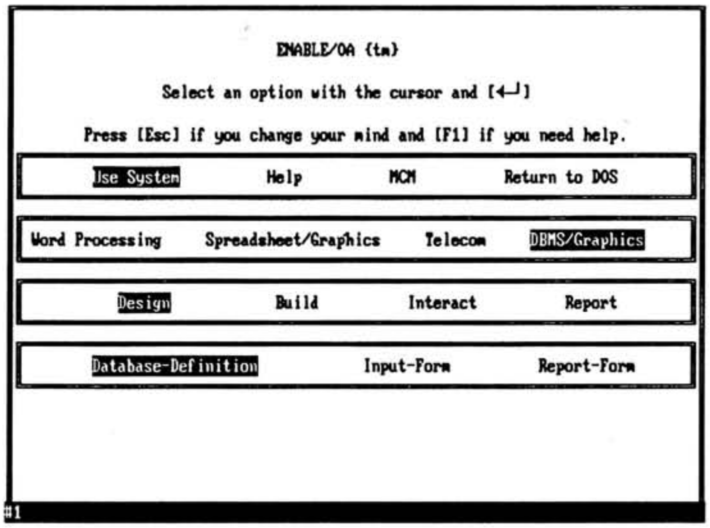
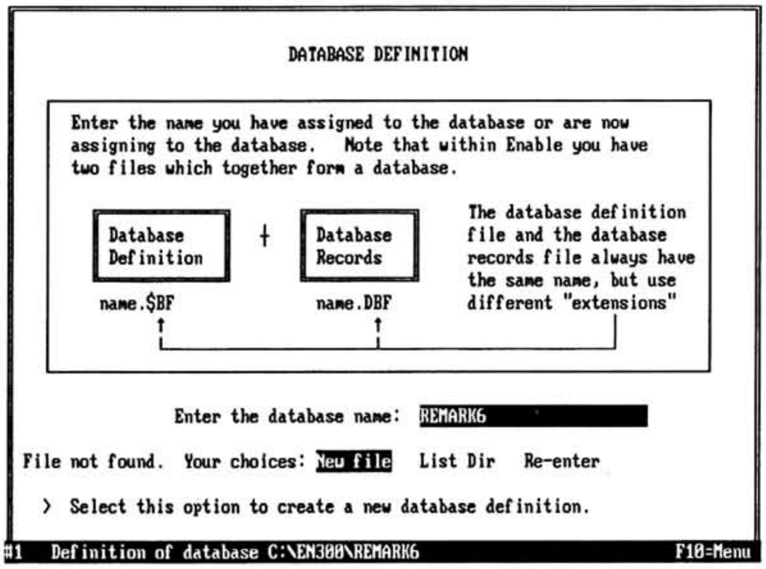


Figure One - Enable Main Menu Screen

patible with dBase. The part that makes Enable powerful is the database definition. This file contains the Enable detailed parameter information about the fields. Many functions that you must program to accomplish in dBase can be easily completed during the construction of the definition file.

After you have designed your database on paper, it is time to move to the keyboard. After entering Enable, select (U)se system, (D)BMS/Graphics, (D)esign, and (D)atabase Definition. Enable will now display the first of the database screens. In this first screen, you are prompted for the name of the database you wish to create or modify.

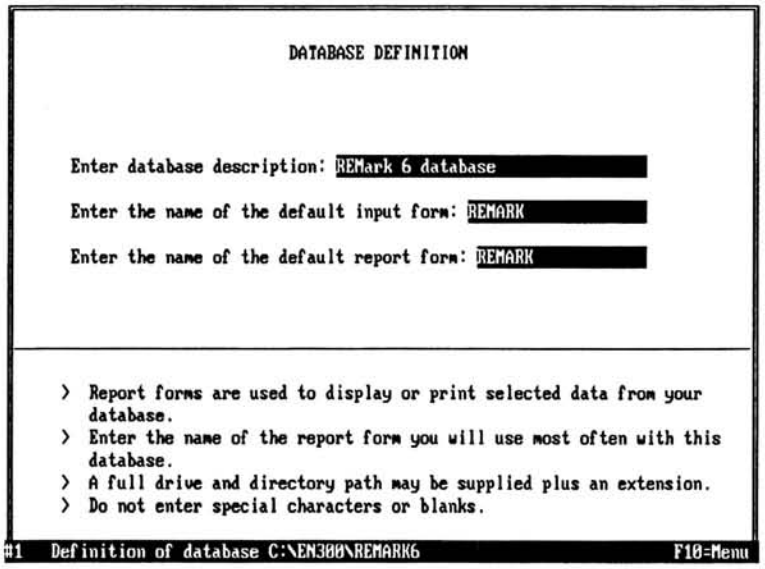
If you are creating a new database, Enable will prompt you to RETYPE or LIST the directory or create a new file. The default is New File. Note that Enable displays information on how it creates database files. You are now on your way to creating a definition that will help during the input of records.



```
                         DATABASE DEFINITION

   Enter the name you have assigned to the database or are now
   assigning to the database.  Note that within Enable you have
   two files which together form a database.

   Database          +   Database          The database definition
   Definition            Records           file and the database
                                           records file always have
   name.$BF              name.DBF          the same name, but use
      ↑                     ↑              different "extensions"

                 Enter the database name: REMARK6

 File not found.  Your choices: New file   List Dir   Re-enter

   > Select this option to create a new database definition.
#1  Definition of database C:\EN300\REMARK6                F10=Menu
```

Figure Two - Definition Screen

After selecting New file, Enable will now move to the second option screen in the definition file. This screen permits you to input a short description. As noted on the bottom of the screen, this is only displayed or printed when the definition is selected. The next option is the name of a default input form. This can be any name you desire although is normally the name of the database. Next is the name of the default report form. Again, this can be any name you wish. When you finish this input, Enable then moves to the screen you will use to create the fields.



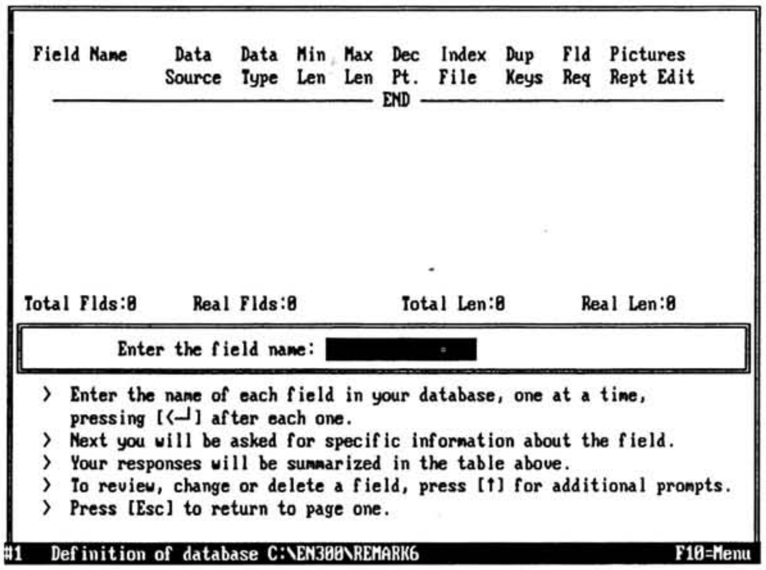
```
                         DATABASE DEFINITION


   Enter database description: REMark 6 database

   Enter the name of the default input form: REMARK

   Enter the name of the default report form: REMARK


   > Report forms are used to display or print selected data from your
     database.
   > Enter the name of the report form you will use most often with this
     database.
   > A full drive and directory path may be supplied plus an extension.
   > Do not enter special characters or blanks.
#1  Definition of database C:\EN300\REMARK6                F10=Menu
```

Figure Three - Screen Two, Database Definitions

Enable permits you to enter up to 254 fields in the database. Each field can contain up to 254 characters. The field name can be up to 10 characters long and contain letters, numbers, UNDERSCORE and COLON.

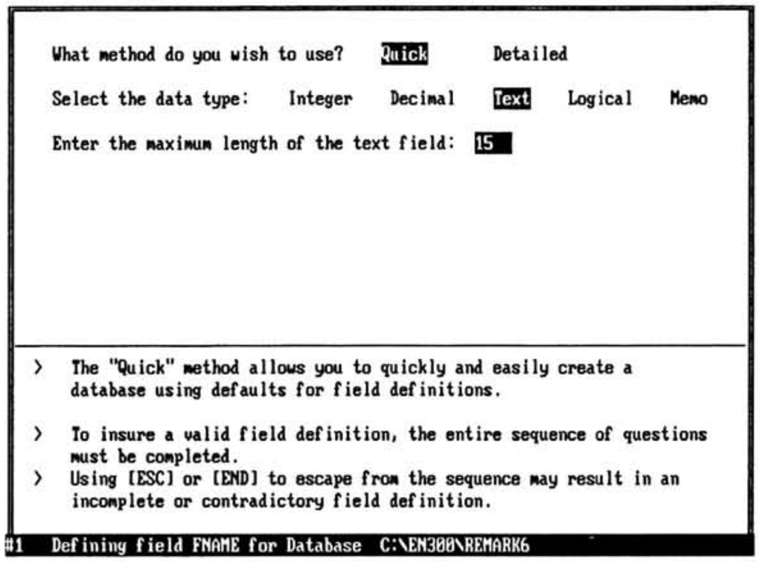
As you enter fields, Enable will maintain a running total of Total Fields, Real Fields, Total Length, and Real Length. We will watch these numbers grow as fields are added. Also displayed on this screen is a summary of the field information. For this example, we will create a simple database that is designed to contain family information.

There are two methods for defining fields, Quick or Detailed. For the first field, the QUICK method will be used. The field will contain the first name of the person. This is a text field. For the name type in FNAME and press (ENTER). Enable will now display the QUICK input screen. The first choice is the QUICK or DETAILED definition selection. Because this field is not indexed and has no special requirements, the QUICK method will meet our needs. After selecting the Quick method, the next option is the type data. You can select Integer or Decimal for numbers. If you select decimal, you



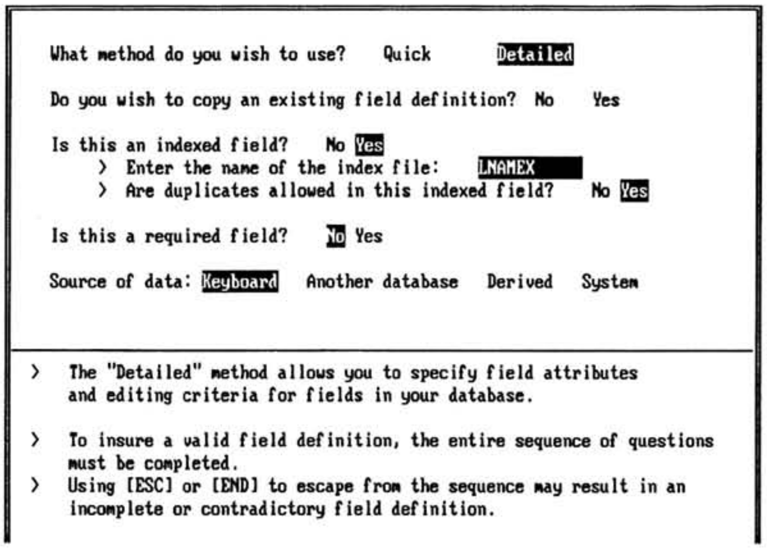
```
  Field Name    Data   Data  Min  Max  Dec  Index  Dup   Fld  Pictures
                Source Type  Len  Len  Pt.  File   Keys  Req  Rept Edit
  ─────────────────────────────── END ──────────────────────────────



 Total Flds:0      Real Flds:0        Total Len:0        Real Len:0
            Enter the field name:
   > Enter the name of each field in your database, one at a time,
     pressing [<┘] after each one.
   > Next you will be asked for specific information about the field.
   > Your responses will be summarized in the table above.
   > To review, change or delete a field, press [↑] for additional prompts.
   > Press [Esc] to return to page one.
#1  Definition of database C:\EN300\REMARK6                F10=Menu
```

Figure Four - Field Definition Screen



```
What method do you wish to use?   [✓ Quick]      Detailed

Select the data type:   Integer   Decimal   [✓ Text]   Logical   Memo

Enter the maximum length of the text field:  [✓ 15]

 >   The "Quick" method allows you to quickly and easily create a
     database using defaults for field definitions.

 >   To insure a valid field definition, the entire sequence of questions
     must be completed.
 >   Using [ESC] or [END] to escape from the sequence may result in an
     incomplete or contradictory field definition.
#1   Defining field FNAME for Database   C:\EN300\REMARK6
```

Figure Five - Quick Field Screen



```
What method do you wish to use?   Quick      [✓ Detailed]

Do you wish to copy an existing field definition?  No     Yes

Is this an indexed field?    No [✓ Yes]
     >  Enter the name of the index file:     [✓ LNAMEX]
     >  Are duplicates allowed in this indexed field?    No [✓ Yes]

Is this a required field?     [✓ No] Yes

Source of data: [✓ Keyboard]   Another database   Derived   System

 >   The "Detailed" method allows you to specify field attributes
     and editing criteria for fields in your database.

 >   To insure a valid field definition, the entire sequence of questions
     must be completed.
 >   Using [ESC] or [END] to escape from the sequence may result in an
     incomplete or contradictory field definition.
```

Figure Six - Screen One, Detailed Report

are then prompted for the number of decimal positions. The text option permits you to specify the number of positions the field will occupy. Logic will prompt for a "Y"es or "T"rue. After responding to the length, Enable will now return to the field summary screen.

The next field to add will be the last name. Because this field will be indexed, the DETAILED definition method must be used. Indexing will permit you to find a record in one second or less. For DETAILED definitions, Enable will display the first of three screens. On the third line, select "Yes" for "indexed." Enable will prompt you for the name of the index file and if duplicate entries are permitted. In most database programs, you would be required to enter this data as a separate program in the input screen. Enable permits you to specify if the field is required and then where the data will come from. See Figure Six for the screen display.

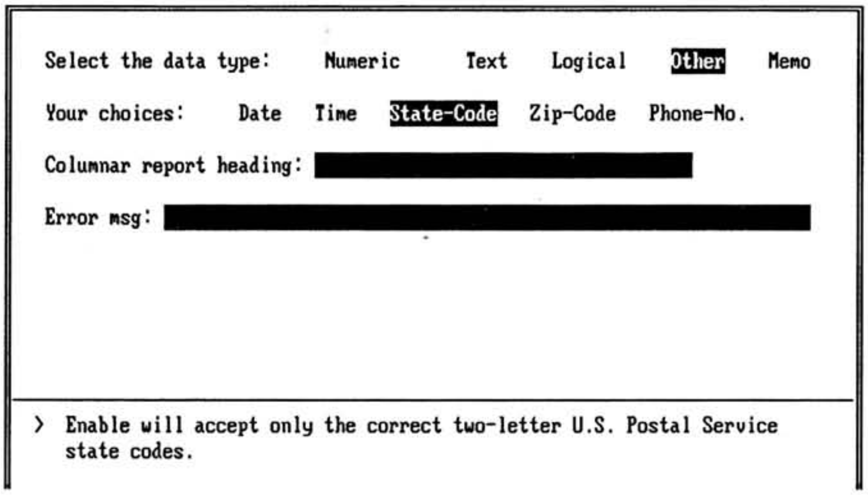
After finishing this screen you are moved to the second screen. For the current field definition you only need enter the type and length of the field. The last screen permits you to define the look of the output in a report and an error message, if desired. Pressing (RETURN) will display the field summary screen again.

Using these procedures, enter the rest of the personnel data. STREET is a quick text field while the CITY can be a detailed field with an index, or quick. The next field is STATE. This is one of the unique field identifiers available in Enable.

For the state definition select detailed. The first page is defined as required. The field may be indexed if required. On the second page of the definition select "Other." Enable will display additional options, one of which is State-Code. The information displayed on the bottom of the screen indicates that Enable will now accept only the two letter State code. You are then prompted for a Columnar heading if you wish to use something other than STATE. The error message block permits you to enter a message that will display on the status line if an incorrect entry is made. As an example, for the state code you could insert "You must enter the two letter postal code".

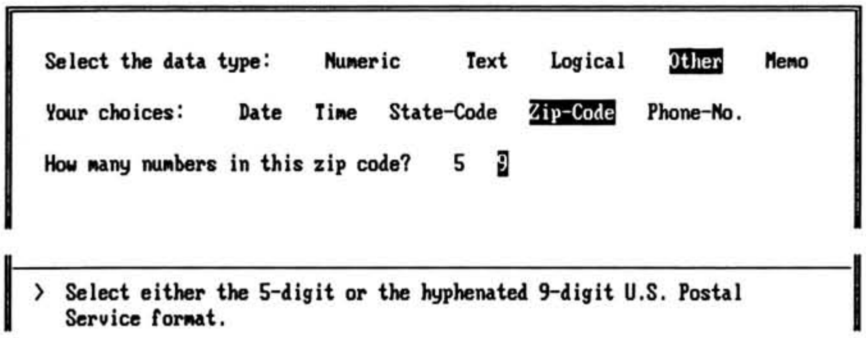
The next field is ZIP code. Again, this is one of Enable's unique field types. Select detailed definition and Other on the second page. Like the State-Code, Enable has a Zip-Code definition. Selecting this option permits you to select either five or nine digit ZIP Codes. See Figure Eight. After ZIP code comes the employee phone number.

From the ZIP code field, you remember that there is a detailed definition for Phone-Number. Selecting this option, Enable will prompt you for the format of the number. See Figure Nine for these options. Enable uses the State-Code, Zip Code and



```
Select the data type:    Numeric    Text    Logical    [✓ Other]    Memo

Your choices:     Date   Time   [✓ State-Code]   Zip-Code   Phone-No.

Columnar report heading:

Error msg:

 >  Enable will accept only the correct two-letter U.S. Postal Service
    state codes.
```

Figure Seven - Detailed State Definition



```
Select the data type:    Numeric     Text    Logical    Other    Memo

Your choices:    Date   Time   State-Code   Zip-Code   Phone-No.

How many numbers in this zip code?    5   9

> Select either the 5-digit or the hyphenated 9-digit U.S. Postal
  Service format.
```

Figure Eight - Zip Code Definition

Area Code of the telephone number. When you enter data in the database, Enable checks the data and will display an error if the ZIP code does not exist in the State. It also checks the area code of the phone number with the State-Code. If the area code does not exist in the State, an error message will be displayed on the Status line of the screen.

The next field in this database is employee number. It is a quick Integer field. The reason for this will be covered later. The last field in this database is the REMARKS field. This is a MEMO field and is selected from the Quick definition menu. The Memo field takes up ten positions in the database, and is used as a pointer to the file with the actual data.

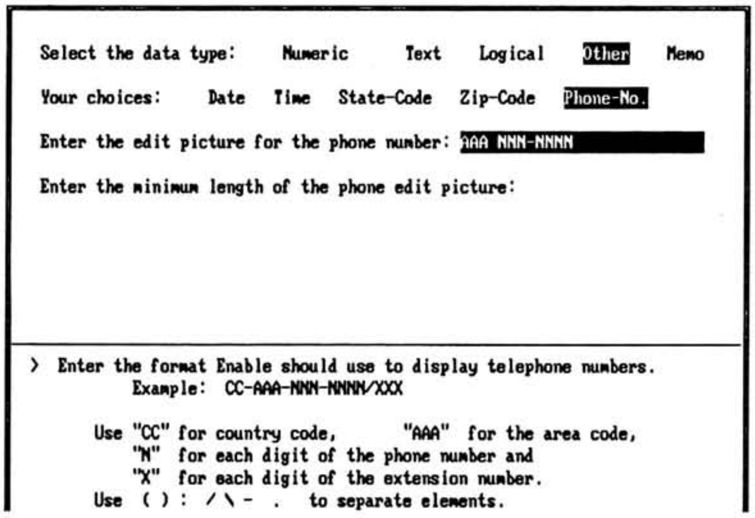
Figure Ten shows the database field summary screen. This screen displays eight fields in the definitions. This screen shows the total number of fields, both real and total, and the lengths. It is possible to rename the fields or redefine the displayed parameters of the fields. To do this simply move to the field, move the highlighted bar to the area you wish to redefine using the right or left cursor keys and press the space bar. Enable will display acceptable values.

After you have completed the database definition, press F10 to display the Top Line menu options. You may now print the database definition from this menu. The printed definition is very complete and is in fact an over kill. Select Save and then press F10 and accept Quit. Enable OA has been improved in this area. Earlier versions of Enable could crash at this point if you had modified the database definition if it already had data in it. Now Enable simply tells you that the definition was changed and asks if you wish to move the old data into the new definition.

Figure 11 is the screen display that Enable presents. If you select the COPY command, Enable has a built-in macro that will generate the new database. At this screen, simply select YES and Enable will start the process. It will create a temporary database which is then copied into the new definition. You are permitted to change the way the data goes into the new definition. Enable will display a screen that matches the old database with the new database. You can change the name in the new database if desired. See Figure 12 for the screen display.

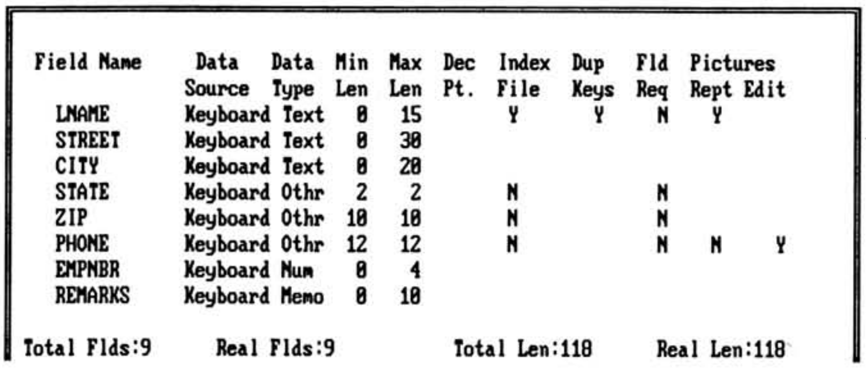
You have now completed a basic Enable database definition file. You can now input data by using the basic capability of Enable. This is accomplished by typing (U)se system, (D)BMS/Graphics, (I)nteract, and (A)dd and typing in the name of the database. In the next article, we will create an input screen that will display data from other databases and create a output report.

Continued to Page 35



```
Select the data type:    Numeric    Text    Logical    Other    Memo

Your choices:    Date   Time   State-Code   Zip-Code   Phone-No.

Enter the edit picture for the phone number: AAA NNN-NNNN

Enter the minimum length of the phone edit picture:

> Enter the format Enable should use to display telephone numbers.
        Example:  CC-AAA-NNN-NNNN/XXX

     Use "CC" for country code,      "AAA"  for the area code,
         "N"  for each digit of the phone number and
         "X"  for each digit of the extension number.
     Use  ( ) :  / \ -  .   to separate elements.
```

Figure Nine - Telephone Definition



```
 Field Name    Data     Data  Min  Max  Dec  Index  Dup   Fld  Pictures
               Source   Type  Len  Len  Pt.  File   Keys  Req  Rept Edit
   LNAME       Keyboard Text   0   15          Y      Y    N    Y
   STREET      Keyboard Text   0   30
   CITY        Keyboard Text   0   20
   STATE       Keyboard Othr   2    2          N           N
   ZIP         Keyboard Othr  10   10          N           N
   PHONE       Keyboard Othr  12   12          N           N    N    Y
   EMPNBR      Keyboard Num    0    4
   REMARKS     Keyboard Memo   0   10

Total Flds:9       Real Flds:9          Total Len:110       Real Len:110
```

Figure Ten - Definition Summary Screen

On the Leading Edge

William M. Adney
P.O. Box 531655
Grand Prairie, TX 75053-1655

More on Memory: UMBs and HMA, System Configuration, Lotus 1-2-3 Upgrade, Speaking of Software Upgrades

I am beginning to receive more and more letters with questions about memory, specifically about some of the "new" terminology that is finding its way into the manuals. To help you understand what some of these new terms (UMBs and HMA) are all about, I have included a complete description of them in this article, as well as a review of the four kinds of memory.

I also recently received an upgrade notice for Lotus 1-2-3 which I found was at least interesting and more than a little disappointing. As you will see, one thing you should always do is read the fine print in upgrade notices and license agreements. Let's begin by first taking a look at what the HMA and UMBs really are.

More on Memory

Given that much new software is using all kinds of memory, you may have noticed that there seem to be all kinds of new names for types of memory, such as HMA and UMBs, if you have been reading about new software. Although there are apparently some new acronyms that refer to memory, it turns out that the memory they describe has actually been around for years. Unfortunately, these terms are beginning to appear in some documentation, such as the ZDS MS-DOS 4.0 User's Reference, and there has been no real description of how these new terms relate to what most users are familiar with. But before we jump into what the HMA and UMBs are, let's review a few things that I have mentioned before, most recently in last May's column.

Memory for 8088-based Computers

For 8088-based computers, there are three kinds of memory. The first is the conventional memory that most users know about, and it is defined as memory between 0 KB and 640 KB. Although this kind of memory is frequently called RAM, that can be confusing because virtually all of today's computers with CPUs ranging from the 80286 on up to the 80486 are advertised as having at least 1 MB of RAM (normally using either 1 MB RAM chips or more often a 1 MB SIMM). More on that when we take a look at memory for 80286 and newer computers.

And because the 8088 has an address space of one megabyte (1024 KB), the memory between 640 KB and 1024 KB is the second kind of memory. It is reserved for the ROMs, such as the system and video ROMs, and RAM for various add-on boards, such as video cards.

This reserved memory was discussed in more detail in my February 1990 column, but it generally works out this way. It is divided into 64 kilobyte "chunks" which are BLOCKS of memory. Each memory block was originally defined and reserved by IBM for some specific purpose. The first block from 640 KB to 704 KB is reserved for EGA video RAM. The second block from 704 KB to 768 KB is reserved for MDA and CGA video. The third block from 768 KB to 832 KB is reserved for miscellaneous expansion, such as the hard drive ROM and EGA ROM. The fourth and fifth blocks from 832 KB to 960 KB were originally reserved by IBM and eventually used for program cartridges for the PC Jr., which explains why the PC Jr. only had slots for two cartridges. And the sixth block from 960 KB to 1024 KB was reserved for the system ROM. Keep in mind that these were the original definitions for the use of memory between 640 KB and 1024 KB, and many of you probably know that the actual usage of these memory blocks has changed in the last few years.

When I mention such things as memory blocks being defined as 64 KB chunks of memory, I usually get at least one letter asking why. The most complete answer involves a rather complex discussion of the evolution of the CPU used, but suffice it to say that it is really based on the address capability of the 8-bit CPUs, such as the 8080 and 8085, which allowed a maximum of 64 KB of memory. When the 8088 was originally developed, it used a 20-bit address bus which can be shown to be limited to the 1 MB address space that I previously mentioned. However, it is much easier to consider the memory addresses of the various blocks of memory in hexadecimal, such as the very first (i.e., lowest) memory block from 0000:0000H to 0000:FFFFH. If one converts FFFFH to decimal, that becomes 65,535, and if you count the "zero" address (which you must do), you have a total of 65,536 possible addresses. Divide that by 1024 bytes per kilobyte, and you will find the result is exactly 64 KB. That's perhaps the easiest explanation as to why 64 KB is a memory block.

Although I strayed away from the discussion about the second kind of memory in an attempt to describe how it works, the important point is that each of these 64 KB chunks of memory from 640 KB to 1024 KB are called *Upper Memory Blocks* or *UMBs*. In other words, the second kind of memory consists of UMBs, which is just the six 64 KB memory blocks mentioned above.

The third kind of memory available on an 8088-based system is expanded memory, which allows one to "expand" memory beyond the original 640 KB conventional memory limit. Expanded memory, frequently called EMS (for Expanded Memory Specification), is based on an add-on board containing RAM chips and must be "initialized" by installing the expanded memory device driver (in CONFIG.SYS) provided by the manufacturer of that board. For 8088-based systems, such as the Z-150 series, one MUST use the expanded memory device driver furnished by the board's manufacturer. And I will again note that the EMM.SYS device driver provided with various versions of ZDS MS-DOS will NOT work for this because I continue to get letters about "bugs" in EMM.SYS when a user finds it does not work with an add-on board in a Z-150 or other 8088-based system. This is NOT a bug, and there is simply no way to get EMM.SYS to work in a Z-150, period.

In summary, there are three possible kinds of memory in an 8088-based computer: conventional memory from 0 KB to 640 KB, UMBs from 640 KB to 1024 KB, and expanded memory which is used to expand memory beyond the original 640 KB limit. Conventional memory and UMBs are an integral part of the design of these systems, and expanded memory must be added if needed. I should also note that adding expanded memory is a waste of money UNLESS you have at least one program (such as VDISK.SYS) or application which can use it effectively.

Memory for 80286 and Newer Computers

The 80286 and newer computers also have the same three kinds of memory that 8088-based computers do, but there is one quirk involving the UMBs for today's computers that have at least 1 MB of contiguous RAM installed using either 1 MB RAM chips or 1 MB or larger SIMMs. That quirk allows you to configure part of the 1 MB of RAM as expanded memory without having to add any additional hardware to your computer as I have mentioned before. On my SupersPort 386SX for example, I can define expanded memory in the SETUP, install the EMM.SYS device driver, and I have about 256 KB of expanded memory that can be accessed by the Quattro Pro 3.0 spreadsheet program that I frequently use. This trick does not work on the Z-241, Z-248, or any of the old IBM ATs because their original conventional memory was 512 KB because 512 Kb RAM chips were installed.

Why does this work on systems with a 1 MB of RAM? Well, the key is that even though older systems had a 1 MB address space, not all of that was Random Access Memory (RAM). As I said earlier, some of that was reserved for ROMs, such as the hard drive ROM and video ROMs. Some of the "extra" RAM (from 640 KB to 1024 KB) available on 1 MB systems can be used for other things, such as expanded memory, so long as there is no address conflict with existing ROMs and video RAM. That's why the amount of expanded memory you can get is limited, but you still get some "free" expanded memory on these current systems. And regardless of what kind of memory we are talking about (either RAM or ROM), the 64 KB chunks of memory from 640 KB to 1024 KB are still called UMBs. When your system actually works with this "upper memory area", one part of a UMB may be a ROM address space, and another part of that same UMB may be a RAM address space. The resolution and arbitration of these memory addresses are defined in the system, although an expanded memory device driver like EMM.SYS figures out how much "free" RAM space is available for its use. For example, the highest UMB (from 960 KB to 1024 KB) is NOT available because it is reserved for the system ROM. And because current Zenith Data Systems computers use additional UMBs to load ROM into RAM (because it's RAM is faster than ROM), that memory space is also not available. ZDS calls the technique of loading ROM into RAM "slushware", although you will occasionally see the term "shadow ROM" which is basically the same thing. Now let's look at the fourth kind of memory that is only available on 80286 and newer computers.

Extended memory. Extended memory is the memory that BEGINS at 1024 KB (1 MB). For that reason, it is not possible to have extended memory on an 8088-based computer because 1024 KB is the address limit for the 8088 CPU. Only 80286 and newer computers can access memory beyond 1 MB. However, you may see a new term which identifies a specific part of extended memory.

The FIRST 64 KB memory block (from 1024 KB to 1088 KB) in extended memory is called the *High Memory Area or HMA*. Although I have occasionally seen references which indicate that HMA is "any memory" above 1 MB, that is not correct; it is only the first memory block. Since many of you may be interested in why, let's digress a moment to look at the reason.

The 80286 CPU has two modes: the Real Mode and the 286 Protected Mode. The Real Mode was intended to EXACTLY emulate the 8088 CPU, which may occasionally be referred to as the "8088 Mode". As a side note, the 8088 CPU has only one mode, which is why you can only run Windows 3.0 in the Real Mode on an 8088-based system. The 286 Protected Mode, frequently referred to as just the Protected Mode, is required to access all of the 80286's advanced features, such as additional memory and other enhancements. But the most interesting part of this discussion centers on how Intel designed the Real Mode emulation for the 80286 CPU.

How you describe the 80286's Real Mode emulation depends on your perspective. One way is to say that the 80286 has a "bug" which does not specifically emulate the 8088. As I mentioned earlier, the 8088 has a 20-bit address bus that allows it to directly address EXACTLY 1 MB of main memory, and any attempt to access memory beyond the 1 MB boundary would cause a wrap-around to the beginning of memory. Despite the fact that Intel designed the 8088 (and the 80286), the 80286 Real Mode did not emulate that wrap-around feature. Unfortunately, that became obvious when IBM released the original 80286 AT computer, and it would not always run software that was designed for the 8088. Without going into all the technical details, suffice it to say that the reason is that some programmers took advantage of the 8088's wrap-around "feature" (apparently to save some program code), which was not available in the 80286 Real Mode.

Even if you don't remember it, perhaps you can imagine the flap that occurred when it was discovered that the new IBM AT was apparently not "software compatible" with the original PC. To be fair to IBM, that was not their fault because programmers took advantage of the 8088's wrap-around feature that Intel's 80286 CPU did not emulate. And to be fair to Intel, I don't really consider the "problem" a bug because there is virtually no way they could have anticipated that programmers would take advantage of such a low-level CPU hardware capability, which was not documented or "approved" by Intel. As I've said before, a programmer is taking a fairly big risk in relying on "undocumented" features or capabilities, especially for hardware, since there is absolutely no guarantee that the next generation of hardware will include those same features or capabilities. This kind of problem is the reason that I do not recommend attempting to exceed the design capabilities of a any computer or software. Unfortunately, I believe that this kind of problem will continue to occur because programmers will attempt to save time (in coding a program) or improve program speed by using hardware features that may or may not be "standard" or "approved". In any case, IBM was forced to find a hardware solution to the apparent problem of 8088-software incompatibility on the AT.

In technical terms, what actually happened was that the 80286 was not always sending out a logical zero signal on the A20 line when it should have been to correctly emulate the 8088 in Real Mode. If the A20 line sends a logical one, it is able to access memory above 1 MB (extended memory), which explains why the A20 line MUST be zero to correctly emulate the 8088. In case you are unfamiliar with the terminology, the "A20 line" is the 20th memory address line, beginning at zero, that is generated by the 80286 and later CPUs.

To fix the problem, IBM had to make a slight change to the AT's design to force the A20 line to a logical zero in Real Mode, even when the 80286 CPU was trying to hold a logical one. That allowed all of the old 8088-based software which used the wrap-around effect to run on the AT.

But there is a more interesting ending to this story: it is the basis for QuarterDeck Office Systems' DESQview multitasking program that I mentioned in a previous column. DESQview was the first program to utilize the HMA by controlling the state of the A20 line, and the company has always claimed they invented the programming technology which made the HMA usable. It's interesting to note that Microsoft finally decided to really use that technology because reports on MS-DOS 5.0 indicate it is taking advantage of this feature for 80286 and newer computers by managing the state of this A20 line. That allows you to use the HMA for DOS itself so long as the CPU is in the Real Mode, assuming that you have some extended memory installed of course. Perhaps you will find that explains why I went into so much detail as to what is going on with the A20 line and the HMA.

As you can see, memory management is a tricky business. If it is not done exactly in the correct way, strange problems occur: programs won't run, the computer freezes, or other unpredictable things can happen. And even if you are not particularly interested in the technical details, I hope that you now understand what the four kinds of memory are, how they are used, and what they can be used for.

Memory from 0 KB to 640 KB is called conventional memory. Memory from 640 KB to 1024 KB (1 MB) consists of six blocks called UMBs. Expanded memory is used to "expand" memory capacity beyond the conventional memory limit (that's a hardware limitation, NOT a DOS limitation) of 640 KB. Extended memory is memory that is above 1 MB, and the HMA is the first 64 KB block of extended memory.

System Configuration

Even if you understand what all these different kinds of memory are and how they are generally used, you may still not be sure how to configure your computer's memory to the best advantage. As usual, I will caution you that it is a waste of money to add memory that your programs cannot use, so be sure to check your software manuals before you buy anything. I will assume that you already have 640 KB of conventional memory.

For systems that have 1 MB of RAM (80286 and later systems), you can get some "free" expanded memory by installing the EMM.SYS device driver in the CONFIG.SYS file as I have mentioned in previous articles. In most cases, you will also have to use the ROM-based SETUP program to configure that memory as expanded (usually shown as EMS on the menu) memory. Whether or not this helps depends on what program you are using. Quattro Pro for example, will automatically detect and use expanded memory if it's available. If you don't have any programs that can use expanded memory, then even setting this up is a waste of time and conventional memory space because the EMM.SYS device driver requires a small amount of conventional memory to map the UMBs.

For 80286 and later systems, it seems best to configure all additional memory above 1 MB as extended memory, regardless of whether you need expanded memory or extended memory. Programs such as Windows 3.0 seem to prefer extended memory, and of course DOS 5.0 apparently can use the HMA to reduce conventional memory requirements and allow you to use that conventional memory space for other programs. That is becoming more important as the DOS itself becomes larger and could prevent the use of an application that requires a lot of conventional memory, especially if you use one or more memory-resident programs (TSRs).

Now if you have an 80286 or later system and need expanded memory, I have found it is easiest to use Quarterdeck's QRAM (for an 80286) or QEMM386 (for an 80386) to "convert" extended memory to expanded memory. If you have a Z-241 or Z-248 with a ZDS memory card (which is for extended memory only), then you will need a product like QRAM if you need expanded memory as I mentioned in the May column. Again, buying additional memory for a computer is a waste of money unless you have a specific program that can use it or requires it, such as Windows 3.0.

For 8088 systems, the only way to add additional memory beyond 640 KB is to buy a third-party expanded memory card, but be sure you have at least one program that can use expanded memory; otherwise it is a waste of time and money.

Lotus 1-2-3 Upgrade

I recently received an upgrade offer in the mail for Lotus 1-2-3 Release 2.3 for DOS. While that may not seem to be particularly newsworthy, there were a couple of things in the offer that caught my attention, especially since I have never owned a copy of Lotus 1-2-3 for DOS. For starters, the address on the mailer was all messed up, and it's a tribute to the Postal Service that the thing even got delivered at all because my business and street names were misspelled to the point that even I had some difficulty recognizing them. As a first impression, that made for a very poor one. It looked like very sloppy work in getting (or translating) a mailing list that was most certainly not part of the normal Lotus software registration.

When I opened the mailer, it offered a "special price" on the upgrade of $119.00 instead of the usual $150.00. That also included a "Free Offer" of SQZ! and Outline in addition to the new features in the release. So far as I could tell, it looked like most of the new features were basically similar to those included in Quattro Pro 3.0 that I reported on last time. It is interesting to note that this version of 1-2-3 also includes a file viewer that allows you to look at a file, but the display of the file viewer looks virtually identical to that provided by Software Bridge and Outside In that I reported on last time. So far, everything is just about what one might expect in an upgrade offer. That is, until one reads the fine print on the upgrade request form itself.

Perhaps I was more than a little curious as to why I received an upgrade offer for software that I don't have, but I read the contents of the mailer quite thoroughly. And I found a couple of what I thought were real surprises in the offer. The first was on the back of the Upgrade Request form under the Mail Order heading which asked: "Please enclose one original 1-2-3 System Disk or the title page of the 1-2-3 Reference Manual for each upgrade ordered." Many manufacturers have the same requirement, so that is no particular surprise until I considered the fact that I did not own a copy of 1-2-3 and obviously had never registered one. And because I do not own a copy of 1-2-3, I was not prepared for the next item.

Outrageous. That's the only word I can think of to describe one of the sentences under the "License Agreement" which reads: "Within 90 days of receiving your upgrade, you must destroy all your remaining 1-2-3 disks, both originals and backups." Surely you jest! (I never jest and don't call me Shirley!). One would have to be a fool to do that with any software because sometimes upgrades don't always work the way you expect, and there are new bugs that prevent you from doing things that you have always been able to do before. For example, a bug in the memory management software in Release 3 prevented 1-2-3 from LOADING and running on virtually all Heath and Zenith Data Systems computers as I reported last year. On the other hand, you really MUST destroy all of the old original disks and backups or you will be in violation of the terms of the license agreement. Looks like a Catch 22 situation to me.

So far as I'm concerned, these kinds of

antics with license agreements are the major reason that I have never been fond of Lotus 1-2-3, aside from the fact that the earliest versions were copy protected so that backups could not be made. And although I have occasionally had to use 1-2-3 when a client has asked for some help, I have always found 1-2-3 is far more difficult for me to use than other spreadsheets, although 1-2-3 experts seem to have no problem. Today, I have found that Quattro Pro is a much better choice in terms of cost, capability, and the Borland "No Nonsense" license agreement requirements.

From a user perspective, I don't think the Lotus license agreement is very realistic or reasonable, and I would not recommend any software that has those kinds of terms. Unfortunately, it is usually not possible to read a license agreement before one buys software, which is the reason I have included this information in this article. Fortunately, these kinds of terms are becoming quite rare since most manufacturers have recognized that users have serious reservations about buying software that have onerous requirements like this. As a matter of practice, I keep all my old versions of production software because I never know when I might need it to read a very old file created with it. And I have had to reload an old version occasionally just to be able to access a file because the file format has changed considerably in the latest versions. For that reason, I do NOT recommend the Lotus 1-2-3 Release 2.3 upgrade; Quattro Pro 3.0 is a far better choice because it is cheaper and has more features, especially in the expanded macro language.

Speaking of Software Upgrades

As I was writing about the restrictive Lotus 1-2-3 upgrade, I also began thinking about another related question. Why is it that many people continue to use "old" (and sometimes inferior) and outdated software? In this sense, I think this applies even to currently available product upgrades, such as the latest Lotus 1-2-3 version I mentioned earlier. It also applies to Windows and Windows' applications, not to mention those of us who stubbornly stay with DOS instead of using OS/2.

Take OS/2 for example. When OS/2 was originally released a few years ago, there was a lot of ink about all the advantages that it had, such as multitasking. Even today, some people think that OS/2 is the answer to a lot of problems despite the fact that there is still very little OS/2-specific software that you can find. In short, it has not been very well supported by the software developers, probably because there has not been much user demand for it. Although a number of recent articles indicate that IBM is going to provide considerably more backing (which means they are going to try to push sales) for OS/2 version 2.0, that will probably not be enough to convince 40 million or so PC users to change over to OS/2. There also seems to be a persistent rumor that IBM will code OS/2 in such a way that it will only run on IBM computers, and that is a stupid and shortsighted point of view if it is true.

What about Windows 3.0? Even though there are reliable reports that millions and millions of copies of Windows 3.0 have been sold, I have yet to see much Windows use by any major company, let alone individual users. True, many companies and users (like me) have purchased Windows 3.0, but buying it isn't enough. It is quite clear to me that Windows still has a number of problems that must be resolved, and even though I have Windows 3.0, I don't use it very much.

What about other software upgrades, such as Lotus 1-2-3 or DOS? In many of the companies I visit, a large number of them are still running a DOS version of 3.1 or 3.2, and more than a few are still using a version 1.x of Lotus 1-2-3. I've even asked a few users why they don't upgrade their 1-2-3 version, and most respond that it's too much trouble and costs too much. Besides, their version does all they need to do. I've even asked some users why they don't change from 1-2-3 to Quattro Pro, which has more features (including a color display) and is cheaper than 1-2-3. The answers I get most often include something like "Quattro Pro is not compatible with 1-2-3" (which is not true), or "it will take too much time to learn that new software" (which I also disagree with). Quattro Pro can read 1-2-3 files (and macros) directly, and if one wants to stay with the 1-2-3 command structure, one can always change the Quattro menus to conform. But for those users who have taken the time to change over to Quattro Pro, I consistently hear them say that the Quattro menus are much easier to use than the ones in 1-2-3.

Many of the letters I receive indicate that a number of members are still using a DOS version 2.x, even though ZDS MS-DOS 4.0 has been available for over a year, not to mention that Microsoft released MS-DOS 5.0 in June.

So why do people continue to use old and outdated software? I think a good part of the answer lies in human nature. We tend to stay with what we feel comfortable with, whether it's an old car, easy chair or computer software. Does cost have something to do with it? Are new features and enhancements important? So far as computer software is concerned, I think there is more to it than that.

Improved technology, whether it's part of software or hardware, is no longer a "guaranteed sale" like it used to be, no matter who the vendor is. Even IBM has had two specific demonstrations of that fact. The first is OS/2 that I mentioned. And the second is the PS/2 series computers with the Micro Channel Architecture (MCA) that was intended to be next step beyond the Industry Standard Architecture (ISA). Compared to original projections, sales of both have been dismal for a variety of reasons, including cost. Indeed, the obvious success of Windows 3.0 may be more due to the fact that it runs under DOS, even though it has some costly hardware requirements.

I think there are several different parts in the answer to this question. The first, as I indicated earlier, is that people tend to resist change. That's part of the old "If it ain't broke, don't fix it" approach, which I personally subscribe to in many, but not all, cases. Many new software features and enhancements simply aren't needed by many users, so there is little incentive to spend the money for an upgrade and go through the hassle of installing it. As most of you have probably noticed, I write about software from the perspective of the new features and enhancements that I have personally found useful, and I frequently ignore things that I don't have any use for. For example, I personally have no need for a word processor that has the capability to create newspaper-like columns, although I recognize that some users do. And I doubt that all that many users do. On the other hand, I have a need for a word processor that can generate a Table of Contents and an Index, which is something that many users do not need. Like most of you, I usually buy software that meets a specific need I have, such as Software Bridge and Outside In that I mentioned in the May issue. While not everyone has a need to convert a file from one format to another, I have noticed that more and more users need some kind of software for that because they may use one set of software at work and another set of software at home.

There is little doubt that another part of the answer is cost versus perceived value. Why spend any money on something just for the sake of having the upgrade when there may be nothing new that you can use in that upgrade? For example, I am currently using ZDS MS-DOS 4.0 with DESQview for multitasking, so why would I upgrade to MS-DOS 5.0 when it is released by Zenith Data Systems. I have found that multitasking generally helps me work faster, so I expect to upgrade to ZDS MS-DOS 5.0 in the hopes that it will work even better than the version 4.0 in combination with DESQview. That also assumes that Zenith Data Systems will continue their previous policy of sending out upgrade notices to registered MS-DOS users with an upgrade price of $49. If that changes, then I will have to reconsider, but I will probably get the upgrade anyway so I can give you a report on what I found. For me, the value of a better approach is worth the cost if it can save time for me in the long run. And of course that means I expect to use it in a "production" mode, which also means the software MUST be reliable. That's another part of the answer.

I use my computer primarily as a tool for making a living. I spend most of my time writing consulting reports, various articles, and books. Software that I use in my production system must be reliable and predictable. I read a number of publications to help me make judgments on what to buy and when to buy it. And when I find a good program that is reliable, you will generally find some comments about it in this column. For me, reliability is key because I just can't afford the time to "recover" from software problems.

One test of reliability that I use is to generally stay away from many x.0 versions of software. Of course that depends on the software vendor. For example, I decided to buy ZDS MS-DOS 4.0 and Borland's Quattro Pro 3.0 because my past experience has indicated both vendors carefully test software before it is released. That's not to say that the software is always free of bugs, but it is reliable. Still, I perform testing on even that software before I use it consistently on my production system, just so I know where any problem areas might be.

Not all vendor's x.0 versions are that reliable. Even though a vendor may be reputable and well known, that is no guarantee of reliability. IBM, for example, has had considerable problems with x.0 releases of PC-DOS. In certain circumstances, both PC-DOS 2.0 and 4.0 would clobber a hard drive, which resulted in considerable problems. As I recall, PC-DOS 2.0 would clobber a hard drive if a BASIC program bombed for some reason. And version 4.0 seemed to have considerable problems with partitioning and accessing high-capacity hard drives. The last time I checked, there were something like six patch disks that were issued by IBM for correction of various problems in PC-DOS 4.0.

Many of you may recall the problem (mentioned in one of my columns last year) that Lotus 1-2-3 version 3.0 did not run on ZDS (laptops and desktops) and AST computers because of a bug in the memory management routine in that version. Lotus blamed that bug on the computer manufacturers, and I have never seen anything printed that acknowledged the problem was really in their memory manager. I think software reliability is key, and I think that's another part of the answer as to why many people don't upgrade software. Problem reports like these tend to cause considerable worry, sometimes about the wrong things. I remember a couple of letters that I received about the 1-2-3 version 3.0 problem because users thought it was a ZDS computer problem instead of a bug in 1-2-3. That's the reason I checked into the problem quite thoroughly before I originally wrote about it.

Inertia (If it ain't broke, don't fix it), cost versus perceived value, and reliability. I think those are the major issues surrounding whether or not users upgrade or buy new software. Perhaps these thoughts will help you decide whether or not you want to upgrade to a new software version or stay with the one you are currently using.

Powering Down

I hope that this article helps answer most of the remaining questions about memory and memory usage. I have tried to make this article complete with respect to the introduction of some of these "new" terms, even though you have seen some of this information before.

For help in solving specific computer problems, be sure to include the exact model number of your system (from the back of the unit or series from the Owner's Manual), the ROM version you are using (use CTRL-ALT-INS to find it, except for the eaZy PC), the DOS version you are using (including both version and BIOS numbers from the VER command), and a list of ALL hardware add-ons (including brand and model number) installed in your computer. The list of hardware add-ons should specifically include memory capacity (either added to an existing board or on any add-on board), all other internal add-on boards (e.g., modem, bus mouse or video card), the brand and model of the CRT monitor you have, and the brand and model of the printer with the type of interface (i.e., serial or parallel) you are using. Also be sure to include a listing of the contents of the AUTOEXEC-.BAT and CONFIG.SYS files unless you have thoroughly checked them out for potential problems (e.g., TSR conflicts). If the problem involves any application software, be sure to include the name and version number of the program you are running when the problem appears.

If you have questions about anything in this column, or about Zenith Data Systems or Heath computers in general, be sure to include a self-addressed, stamped envelope (business size preferred) if you would like a personal reply to your question, suggestion, comment or request.

Products Discussed

Powering Up (885-4604) $12.00
Zenith Users' Group
P.O. Box 217
Benton Harbor, MI 49022-0217
(616) 982-3463 (ZUG Software only)

Quattro Pro 3.0 $247.50
Borland International
4585 Scotts Valley Drive
Scotts Valley, CA 95066
(800) 255-8008 (Except California)
(800) 742-1133 (California only)
(800) 237-1136 (Canada only)

Manifest	$60.00
QRAM	80.00
QEMM386	100.00
DESQview	130.00
DESQview386	220.00

Quarterdeck Office Systems
150 Pico Boulevard
Santa Monica, CA 90405
(213) 392-9701

"What Did I Change?"

Answer that question by using Directory Compare. Compare date/size of files in any two directories. Public Domain (source included). Send only $5.00 shipping & handling. For PC or Z100 (specify diskette size)

"Can I Improve My Screen Prints?"

Yes - with **EGAD** Graphics & Text Screen Print PC's and now also for the Z-100 series.

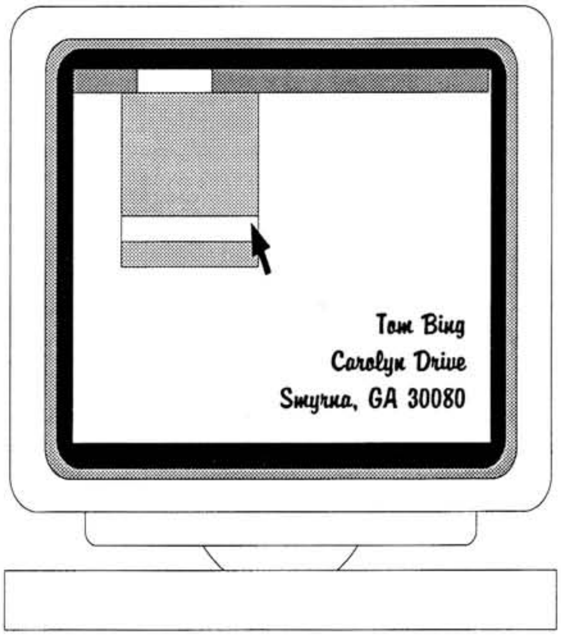
- Prints *IN COLOR* on color printers, or uses black, white and *six gray tones* on most other printers
- Print *any part* of the screen - crop box pops up when Shift-PrtSc (PC) or Shift-F12 (Z-100) pressed; use arrow keys to select region.
- Enlarge graphics 1-4 times
- Supports Super VGA, VGA, EGA, and CGA
- SET program selects printer colors, other options.
- Supports most dot matrix, laser, and ink jet printers (including Epson, NEC-8023, MPI, Okidata, HP, etc.)

EGAD for PC's, **Order # 270 $35.00 postpaid.** Specify 3.5" or 5.25" diskette.
EGAD for Z-100, **Order # 271 $35.00 postpaid.**

LS Software (formerly Lindley Systems)
8139 E. Mawson Rd., Mesa AZ 85207
(602) 380-9175. Call/Write for Free Catalog

Learning C by Computer

Part 2: Learn C Now



Tom Bing
Carolyn Drive
Smyrna, GA 30080

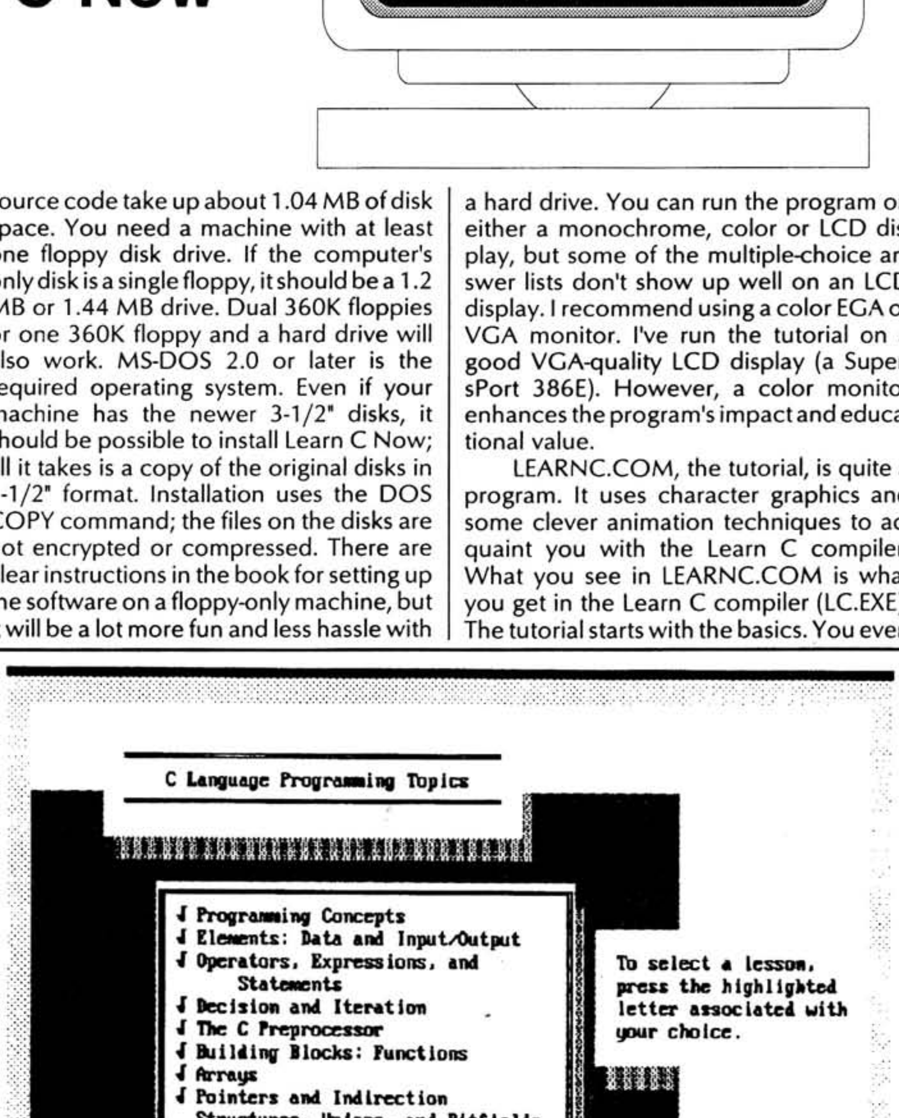
So you want to learn C programming? And the Microsoft C compiler is your choice (or your boss')? What's more, you think learning would be easier if you had a real compiler to experiment with? Take heart! Has Microsoft got a book for you. It's titled "Learn C Now", by Augie Hansen, and it's published by Microsoft Press. The book is more than a long commercial for Microsoft C; it will enable you to get started in almost any C environment. The compiler supplied on disk with the book has its limitations, chiefly that it will not save compiled programs to disk. Nevertheless, it's a good tutorial on the C language, in general, and a good introduction to the Microsoft line of C compilers, in particular.

Part 1 of this series covered the Master C book and disk set from the Waite Group Press. You can run the Master C tutorial program without referring to the book at all. Learn C Now has a different approach; the book and the software are about equally important in learning the material. Its tutorial is in a program called LEARNC.COM, and it's a good introducion to all the book topics except one: graphics programming. Still, you're cheating yourself if you rely totally on the program and never open the book. For instance, the book explains the function of the example programs supplied with the compiler. It also suggests program changes which will shed light on debugging and program logic.

What You Get

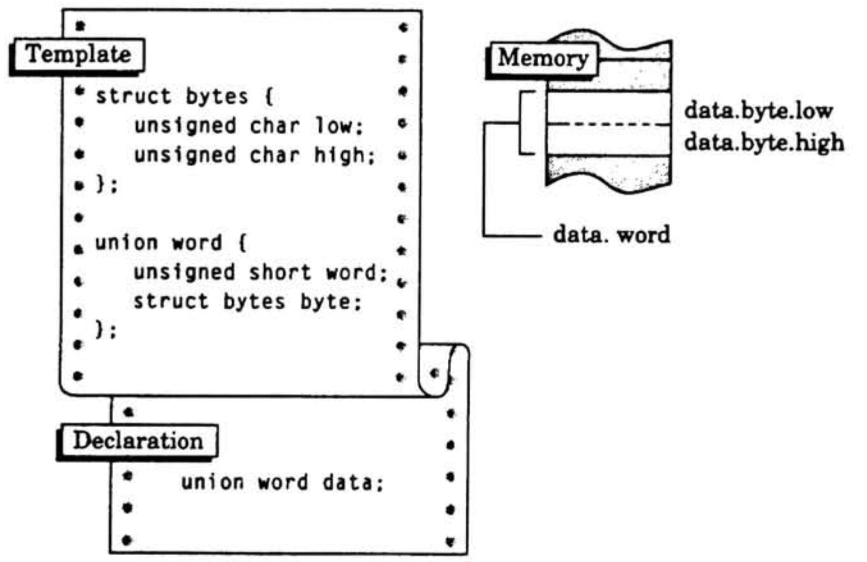
The book is a 367-page paperback. Three 5-1/4", 360K disks are sealed in a pouch in the back cover. The software consists of the Learn C compiler, LC.EXE, the LEARNC.COM program mentioned earlier, header files, and example C programs. I'll refer to LEARNC.COM as "the tutorial" in this article. All the programs and source code take up about 1.04 MB of disk space. You need a machine with at least one floppy disk drive. If the computer's only disk is a single floppy, it should be a 1.2 MB or 1.44 MB drive. Dual 360K floppies or one 360K floppy and a hard drive will also work. MS-DOS 2.0 or later is the required operating system. Even if your machine has the newer 3-1/2" disks, it should be possible to install Learn C Now; all it takes is a copy of the original disks in 3-1/2" format. Installation uses the DOS COPY command; the files on the disks are not encrypted or compressed. There are clear instructions in the book for setting up the software on a floppy-only machine, but it will be a lot more fun and less hassle with a hard drive. You can run the program on either a monochrome, color or LCD display, but some of the multiple-choice answer lists don't show up well on an LCD display. I recommend using a color EGA or VGA monitor. I've run the tutorial on a good VGA-quality LCD display (a SupersPort 386E). However, a color monitor enhances the program's impact and educational value.

LEARNC.COM, the tutorial, is quite a program. It uses character graphics and some clever animation techniques to acquaint you with the Learn C compiler. What you see in LEARNC.COM is what you get in the Learn C compiler (LC.EXE). The tutorial starts with the basics. You even



```
C Language Programming Topics

√ Programming Concepts
√ Elements: Data and Input/Output
√ Operators, Expressions, and
      Statements
√ Decision and Iteration
√ The C Preprocessor
√ Building Blocks: Functions
√ Arrays
√ Pointers and Indirection
  Structures, Unions, and Bitfields
  File Input and Output

  Return to the Main Menu
  Quit (return to DOS)

To select a lesson,
press the highlighted
letter associated with
your choice.
```

Figure 1. Programming Topics Menu.



```
struct bytes {
    unsigned char low;
    unsigned char high;
};

union word {
    unsigned short word;
    struct bytes byte;
};
```

```
    union word data;
```

Figure 2. Sample Illustration from "Learn C Now" Book.

get a tour of the keyboard, if you wish; the tutorial explaining the use of each option. The editor features of the compiler are described thoroughly enough to give even a novice computer user confidence. The tutorial understands two kinds of keyboard input, which I call "navigation" and "control". Navigation means making selections to work through a lesson in the order in which it is presented. For instance, the program will prompt you to press a key and then show you the resulting action taken by the compiler. It usually uses the spacebar to move to the next step in a lesson, or a single letter to select a specific lesson from the main menu. The other kind of input is control, or "Let's do something else". By pressing the 'Ctrl' key along with one other key, the user can either step back through earlier screens, return to the main menu, or exit to DOS before the current lesson is finished.

There is also a lesson titled "Overview of C Programming", giving a summary of the C language (reasons for using it, etc.) and detailing some limitations of the Learn C compiler — for instance, it can't create .OBJ files or place compiled programs anywhere but in memory. It also has to have all components of a C source program (except header files) in a single file. Full-feature compilers don't have these limitations; but then, most of them aren't packaged with a $39.95 book.

The "C Programming Lessons" selection contains a menu of topics which correspond to the chapters in the book (Figure 1). Chapter 13 on graphics programming does not have a counterpart in the tutorial; there are, however, sample graphics programs to run. The "Pointers and Indirection" lesson in the tutorial corresponds to Chapters 9 and 10 of the book. The book and tutorial go hand-in-hand; the book is a useful reference, but its contents really "come alive" when you load and run the example programs with the Learn C compiler.

The book is a well-designed text. Contrasting type fonts distinguish descriptive material from sample programs. The clear black and white illustrations are simple and easy to read, illuminating the difficult points without getting in the way. Figure 2 is a typical example. The depictions of program screens are very legible, appearing in black letters on a white background. This style is quite different from screen displays in the Master C book, which appear as hard-to-read photos of PC screens.

Chapters 2 through 13 usually begin with a general description or definition of the topic, followed by examples in the form of C statements or entire programs. Areas where special care is required are explained thoroughly. Chapters 2 through 12 all end with questions on the material. Six appendices provide details on the following topics:

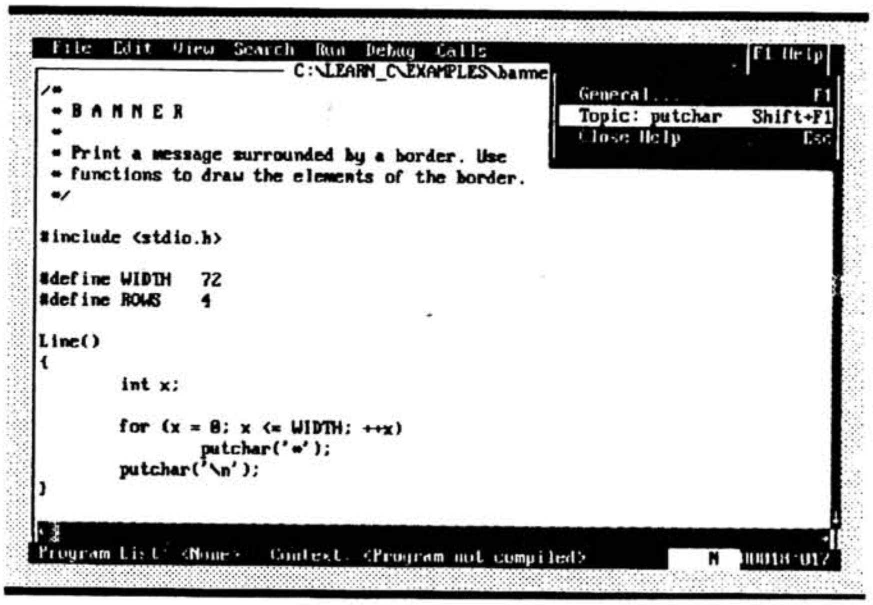
A: C Language Keywords
B: C Language Operators
C: Preprocessor Directives and Pragmas
D: Learn C Standard Library
E: Characters and Attributes
F: C Programming Traps and Tips

Appendix F is followed by a glossary, answers to selected questions and exercises, and a very detailed index.

Chapter Highlights

The following summary attempts to tell what is noteworthy about each chapter. Chapter 1 describes the "look and feel" of the compiler and lists each command used by the editor. Chapter 2 shows how to break down a process to be programmed into simple tasks and introduces an elementary C program. Chapter 3 tells how objects in memory are stored, accessed, and referenced. Chapter 4 clearly explains arithmetic and logical operators and their precedence, i.e., the order in which the compiler processes them in a statement. Chapter 5 covers *if, for, while*, and other program keywords that control program flow, with examples and explanations of common errors. Chapter 6 covers the C preprocessor, citing examples and pitfalls of macro definition.

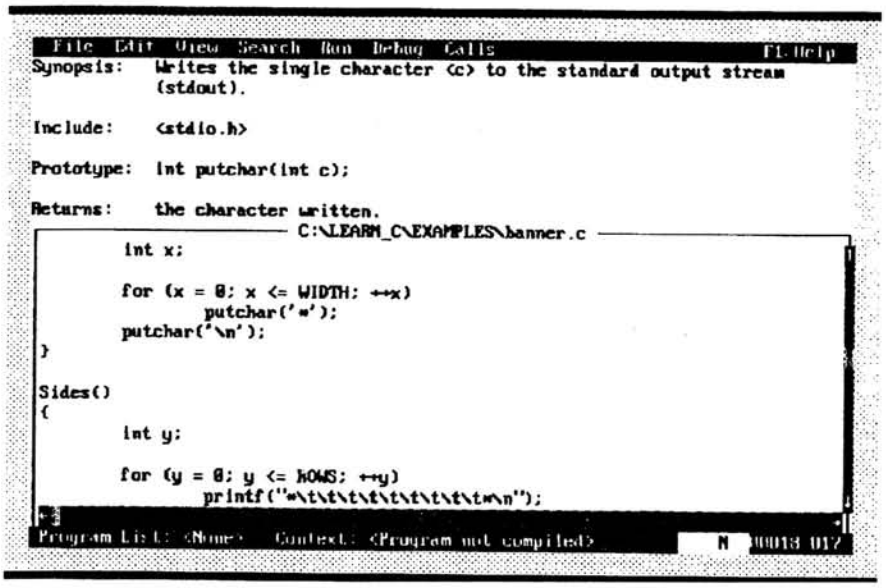
Functions are the heart of C programming. The coverage of functions in Chapter 7 is the core of this book. "Call by value" is used when you don't want to change the values of variables in the calling program; otherwise, "call by reference" is used. The



```
C:\LEARN_C\EXAMPLES\banne
/*
 * B A N N E R
 *
 * Print a message surrounded by a border. Use
 * functions to draw the elements of the border.
 */

#include <stdio.h>

#define WIDTH   72
#define ROWS    4

Line()
{
        int x;

        for (x = 0; x <= WIDTH; ++x)
                putchar('*');
        putchar('\n');
}
```

```
Program List: <None>   Context: <Program not compiled>        N  00018:017
```

Figure 3. Selecting the Help Menu Option.


File Edit View Search Run Debug Calls
F1=Help
Program List: <None> Context: <Program not compiled>
N 00018 017

```
Synopsis:   Writes the single character <c> to the standard output stream
            (stdout).

Include:    <stdio.h>

Prototype:  int putchar(int c);

Returns:    the character written.
                    C:\LEARN_C\EXAMPLES\banner.c
        int x;

        for (x = 0; x <= WIDTH; ++x)
                putchar('*');
        putchar('\n');
}

Sides()
{
        int y;

        for (y = 0; y <= ROWS; ++y)
                printf("*\t\t\t\t\t\t\t\t*\n");
```

Figure 4. Results of Help Selection.

book and the tutorial clearly distinguish between these two concepts, saving the beginning programmer a lot of grief.

Chapter 8 covers arrays, particularly character arrays and strings, and shows how they are initialized. The idea of pointers is a very difficult topic in C, well worth the extended coverage in Chapters 9 and 10. The relationship between pointers, arrays, and strings is shown by appropriate examples. Pointers are shown being used as function arguments in order to return more than one value to the calling program. There is an ingenious example that walks the reader through a simple addition problem using both simple integers and pointers. Chapter 11 uses some real-world programming problems to explain structures, unions, and bitfields. The idea of a union is admittedly an exotic concept for a programmer who has come from a language that doesn't allow that sort of thing. Unions allow you to access an area of memory as either an integer, a floating-point number, a character array, etc. A typical use of a union is to determine the byte order used by your computer to represent integers. To me, that would have been a more realistic example than the UNION_2 program given in the book. However, it might also have been harder for beginning programmers to understand.

Chapter 12 on file I/O explains how text and binary files are distinguished in MS-DOS and the use of the Ctrl-Z as an end-of-file indicator in text files. There is also a discussion of file access types (read, write, append, etc.) and the way these are used with new or existing files.

Chapter 13 on graphics describes some of the library routines used to write to the screen on a pixel (rather than character) basis. It's useful as a way of showing how the C graphics library recognizes the features and limitations of video hardware. Graphics is one area not covered by the Master C package featured in the previous article. However, I believe the professional C programmer would be well advised to use a tested and proven off-the-shelf library of graphics routines, especially if standard (CGA, EGA, or VGA) video displays are to be used.

The 'Debug' Menu Option

The advanced debugging capabilities of the Microsoft and Borland C compilers have greatly lightened the programmer's load in finding and fixing program errors. In the olden days, programmers would add "printf" statements at each point in the program where the value of a variable was to be displayed. This technique is time-consuming, since all those diagnostic *printf*s have to be taken out when the problem is fixed. Also, since code is being added, there is a tendency to create new errors as well as pinpoint old ones. The choices on the Debug submenu point to a better method; the programmer can name variables whose values are to be displayed and specify the statements at which they are to be printed out, but without adding any source code statements. There is an excellent example (pages 206-210) of the use of the Debug option to fix a problem in the 'reverse.c' program supplied on the disks. Even aggregate data objects like structures can be displayed, as well as integer or floating-point variables. Programmers who experiment with Debug in the Learn C compiler will not be intimidated by the Codeview debugger in Microsoft C which inspired it.

On-Line Help

Pressing F1 or Alt-H displays the Help submenu. At this point, either general or topical help can be chosen. The general help screens show C operator precedence, an extended ASCII chart, and so on. The topical screens let the user specify a keyword, such as a function name or C reserved word. For instance, if the user has opened the "banner.c" program for editing or compiling, he or she may need a description of the "putchar" function. The user places the cursor in the word "putchar" on the screen and presses F1. This causes the "Help" submenu to come up with "putchar" already entered in the "Topic" field (Figure 3). The help system hunts up its reference to that topic and prints a condensed definition of the term just below the menu bar (Figure 4). The help screens in Master C are elaborate and provide a detailed reference for students working through the Master C lessons. The help screens for Learn C Now are simpler and more like what is available on a "real" C compiler.

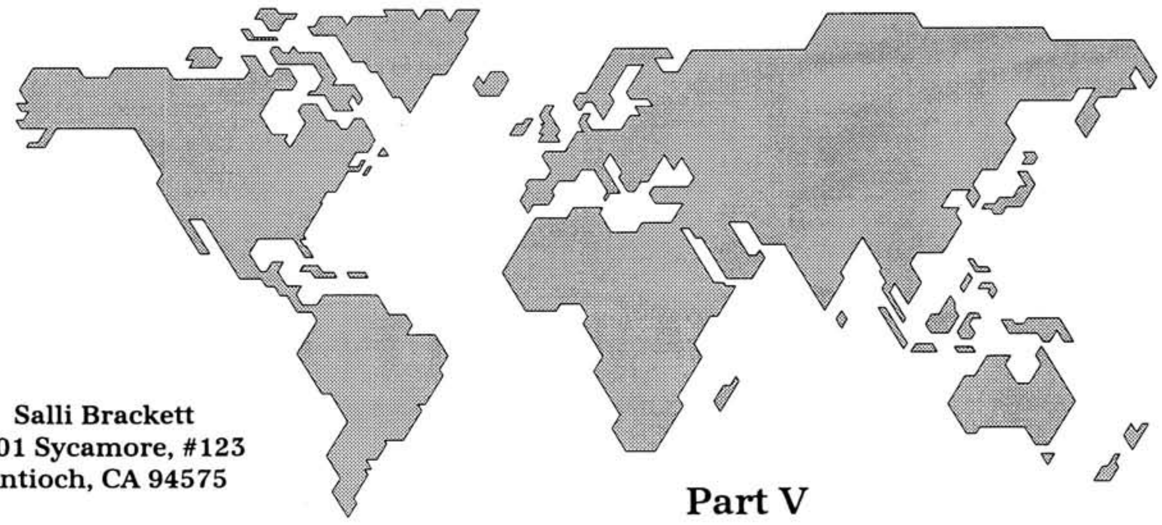
Master C or Learn C Now?

Choosing between these two is a tough call. Unless you're chomping at the bit to go ahead and learn C++, either one of them will be a good introduction to the C language. Master C came out in 1990, after the ANSI C standard was finalized; in Learn C Now, it is referred to as the "proposed" ANSI standard, because Learn C Now was published in 1988. Consequently, Master C has a little more detail on ANSI changes. Master C has a more sophisticated question-and-answer technique. Learn C Now simply explains why a choice was right or wrong. Master C attempts to help a student who is close to the right answer to zero in on it, requiring him or her to repeat material not yet mastered. However, Learn C Now is closer in "look and feel" to what the aspiring Turbo C or Microsoft C programmer will be using. Competitive marketing of these two teaching packages is making the choice even harder; some Learn C Now books come with a discount offer to encourage you to buy Quick C, and Mix Software is offering Master C bundled with their C compiler. The Master C book says that if you already have Quick C, you can install Master C as one of Quick C's menu options. If you find thick textbooks intimidating, and you learn better from computer courseware that can be used by itself, perhaps Master C is for you. If you know you will be using a Microsoft C compiler, Learn C Now will help you ease into that environment. Each of these two packages has helped me solidify my C knowledge.

What Next?

Part 3 in this series will cover the Turbo C++ Disk Tutor offered by Borland/Osborne/McGraw-Hill. Keep watching this space. ✻

The World of WP5.0 and Its Wonders



Salli Brackett
2201 Sycamore, #123
Antioch, CA 94575

Part V

The tab function seems to give a lot of people fits. This article is designed to dissolve the mystery surrounding the tab function. The article is in three parts: an explanation of the tab edit menu, tips on setting up several tab stops for data involving numbers, and a sample table of numbers for input.

When you start working with tabs, you feel that you have really gone 'down the rabbit hole with Alice'. First of all, think of the tab ruler line in the edit menu (sample below) as the ruler line on a typewriter. To set tabs on a typewriter, you space to the position you want and press a tab set key. In the same manner, you can place the cursor on the tab ruler line and type a letter. To make it simpler yet, just type in the number of the column you wish and press ENTER. The advantage to WordPerfect's tab ruler line is that you have choices of types of tabs.

```
              L     L     L
 . . . . . . . . . . . . . . . .
 0     .     10    .     20    .
```

Left Tabs — Standard tabs. Justifies on the left of the tab stop (L).

Decimal Tabs — Tabs center on a decimal (D).

Center Tabs — You can place a center tab in the tab edit ruler line (C). (You also can use the left tab and then center — Shift F6.)

Dot Leader Tabs — Tabs that put dots in front of the tab stop. Type in the number of columns you wish, press ENTER, type a period (.). The L on the column will highlight.

Right Tabs — Justifies on the right of the tab stop (R).

If you wish tabs set at equal intervals: Type <the starting column>, <the interval space> (i.e., 12,4). This means the first tab starts on column 12, and the interval between each tab is 4 columns (spaces). The default is 0,5.

To use the method in this article, you need to change the 'Pos' display on your status line to units of measure (columns and lines). Press Shift-F1, u (in WP5.1, press e, u), d, u, s, u, F7. When using the units of measure method, keep in mind that with the default of Courier 10 pitch, on 8-½" x 11" paper, there are 85 characters (columns), 66 lines, and one inch = 10 characters. These figures will change as you use different fonts. To determine the correct figures with different fonts, see the last line in the Tab Math Table. Once you have these figures, place them on a post-it and put it on your computer for easy reference.

All these figures may seem like a lot to memorize, but try calculated tabs or figuring out what the equivalent of 1 line is in 1". You end up working with the decimal form of inches. For tabbing, you can not use the math method.

The sample for our practice is on the next page. I calculated these tab stops by using the process described in the Tab Math Table. Since this sample needs decimal tabs in columns 2-5, I needed an added calculation for the numbers to line up properly.

Take the longest number and count the digits to the left of the decimal, including commas, spaces and the $ sign. In the above sample, columns 2-5 have 6 characters to the left of the decimal (if there is no decimal, the decimal is assumed at the end of each number). Now you want to add this number (6) to the column number created from your calculations.

Before starting the following exercise, let's calculate our columns based on the

Tab Stop	Sample A	Sample B
1	The beginning of the column 1 is the left margin (11) + the length of the column (20) + spaces between columns (4). 11 + 20 + 4 = 35	The length of the column (20) + the spaces between column (4). 20 + 4 = 24
2	The 1st tab stop (35) + length of column (9) + spaces between columns (4). 35 + 9 + 4 = 48	The 1st tab stop (24) + length of column (9) + spaces between columns (4). 24 + 9 + 4 = 37
3	The 2nd tab stop (48) + length of column (9) + spaces between columns (4). 48 + 9 + 4 = 61	The 2nd tab stop (37) + length of column (9) + spaces between columns (4). 37 + 9 + 4 = 50
4	The 3rd tab stop (61) + length of column (9) + spaces between columns (4). 61 + 9 + 4 = 74	The 3rd tab stop (50) + length of column (() + spaces between columns (4). 50 _ 9 + 4 = 63

Figure 1

Tab Math Table. This may seem like a tedious exercise, but trust me, it is a foolproof method of spacing each column properly the first time. No more guessing or adjusting. The example is based on Helvetica 12 pt type.

1. The longest entry in column 1 is 20 characters and the other 4 columns need 9 characters each.
2. The total is 56.
3. The l/r margins are 11 and the width of the line is 94. 94 - 22 = 72.
4. Results of #3 are 72 and results of #2 are 56. 72 - 56 = 16.
5. There are 5 columns, therefore divide result of #4 by 4, 16/4 = 4.
6. Now we figure the tab stops. Sample A is WP50 (or absolute Tabs in WP51) and Sample B is WP51 relative tabs (see Figure 1).
7. A double check: Remember I said this was foolproof. You can check your math by the following method. Take the last tab stop and add the width of the column. In our example the width of the last column is 9. In Sample A, the result should be the right margin, 83. (The right margin is obtained by subtracting the width of the margin (11) from the total length of the line (94). In Sample B, the result should be the same. Since you didn't use the left margin in your calculation, add it now. 63 + 9 + 11 = 83. I recommend using a calculator even for the simple math. This will save you having to redo the exercise due to a mistake in arithmetic.
8. Remember earlier, I mentioned how to calculate for decimal tabs. Since, in this case, we need decimal tabs for columns 2-5, you need to add 6 to each tab stop (6 being the total number of characters to the left of the decimal).

Therefore:

Tab Stops	Sample A	Sample B
1	35 + 6 = 41	24 + 6 = 30
2	48 + 6 = 54	37 + 6 = 43
3	61 + 6 = 67	50 + 6 = 56
4	74 + 6 = 80	63 + 6 = 69

Be sure and change your units to 4.2 units, have a clear screen, press RETURN four times and begin the exercise. If you are using WP51 and want to use Relative Tabs, in the Tab Edit Menu, press t(ype) and select Relative Tabs.

1. Press Shift-F8, l(ine), t(ab).
2. Cursor to the 0 column on the tab line.
3. Press Ctrl-End (EOL key) to delete existing tabs.
4. Type in the first tab stop as in #8 above, then type d for decimal tab.
5. Repeat number 4 for all the numbers.
6. After completing number 5, Press F7 twice.
7. Type in the columns as shown in Figure 2 using the TAB key.
8. You now need different tabs for your headings. To copy the tab set code to the beginning of the document, do the following:

   Place the cursor on tab set code (with reveal codes, the tab set code should be highlighted), press DEL, y, F1, r. (You just replaced the code you deleted, but the code now can be restored at another position in the document.)

   Place the cursor at the top of the document, press F1, r. (You now have a duplicate of the original tab code. You can now edit this code.

   Make sure the cursor is to the right of the tab set code. Go to the tab set menu (Shift-F8, l,(line) t(ab)), replace the D's with c for center tab and press F7 twice.
9. Type in the headings: Product, January, March, and Total.
10. Remember to put the tabs back to the default (0,5) immediately after your table.

TIP: If you use tab tables a great deal, I recommend creating a macro for the default tabs.

Your finished product should look like the one in Figure 3. Following are some added features for using tabs.

Product	January	February	March	Total
Apple Pie	5344.25	5487.65	6001.50	16,833.40
Bear Claw	2567.80	2010.36	1989.50	6,567.66
Chocolate Eclair	3558.22	-500.01	2775.12	5,833.33
Linertore	1783.15	776.75	765.43	3,325.33
Strawberry ShortCake	981.00	865.40	115.50	1,961.90
Truffled Delight	3229.25	3005.22	3004.04	9,238.51
TOTAL	17,463.67	11,645.37	14,651.09	43,760.13

Figure 3

Apple Pie	5344.25	5487.65	6001.50	16,833.40
Bear Claw	2567.80	2010.36	1989.50	6,567.66
Chocolate Eclair	3558.22	-500.01	2775.12	5,833.33
Linertore	1783.15	776.75	765.43	3,325.33
Strawberry ShortCake	981.00	865.40	115.50	1,961.90
Truffled Delight	3229.25	3005.22	3004.04	9,238.51
TOTAL	17,463.67	11,645.37	14,651.09	43,760.13

Figure 2

Underline Spaces/Tabs

You can have the underline key work on the spaces between tabs. Press Shift-F8, o(ther), u(nderline). Type y, y. Press F7.

<u>This is a heading   heading two</u>

The above was created changing underline/tabs as above. Press underline, type first heading, tab, type second heading, tab, type third.

The default works without underlining between tab stops.

<u>This is a heading</u>   <u>heading two</u>

Align Tab (Ctrl F6)

The align tab key uses the present tab settings. This key can be used with all numbers (ones with decimal points or not). You do not need to set special tabs.

January	$ 5,620	January	$5,620.00
February	265	February	265.00
March	3,623	March	3,623.00

Figure 4

Following the sample in Figure 4, tab over 3 or 4 tabs and type in 'January'. Tab Align (CTRL F6) several times and type in the first number of your list. The numbers will all move to the left. To line up the successive numbers with the first, press tab align until the cursor lines up with the decimal point or the space after the last digit in the number.

If a tab align is not lined up, delete all the tab aligns on that line and re-enter them.

If you wish the numbers to line up on the comma instead of the decimal:

Press Shift-F8, o(ther), d(ecimal). Type , (comma). Press F7 twice.

January	300,000
February	2500,000
March	256,000

To put an underline under the last number in a column:

Tab (the tab key) to just short of the column in question. Space over to the point you wish the underline to start. Press underline (F8). Using the spacebar, create your underline. Under the ^F8 menu, a(ppearance), there is also a double underline.

I do hope this has eliminated some of the mystery of tabs. If you have any specific questions or problems, don't hesitate to write me. Happy tabbing!!

If you are using a 12 pitch font, I recommend that you change the default to 12, 5. Explanation: For a standard 1" left margin with a 12 pitch font, the text starts on column 12. If you have tabs every 5, starting at 0, the second tab is 10, the third is 15. Therefore, with a starting margin of 12, your first tab should be 17, not 15.

5.1 has solved this problem with the use of *relative* (relative to the margin) and *absolute* (absolute to the edge of the paper) tabs. When using the default (relative) tabs, the tab stops change with the margins. So, if the left-hand margin is 10, the tabs are 5, 10, 15, etc. If the left-margin is 12, the tab stops are 12, 17, 22, etc.

If you determine your table stops by eyeballing on the screen, not using the math method described in the Tab Math Table, you must remember to subtract the size of the left-hand margin to each tab stop.

For example, if your tab positions from the screen were 25 and 42 with a 12 pitch font, you would have to make tab stops of 13 and 30; and using 10 pitch, the tab stop would be 15 and 32.

Suggestion 1

If you have text with indented paragraphs, such as in a text with items like 1., 2., A., or B., to get the proper spacing with the indent key, change the space interval of your tabs to 4.

Suggestion 2

1. Determine the length of each column:
   1st column = 4; 2nd column = 30; 3rd column = 12
2. Add length of columns: 4 + 30 + 12 = 46
3. On an 8-1/2" x 11" piece of paper, the length of the paper is:

10 pitch	12 pitch	12 pitch (proportional)
85 (spaces)	102 (spaces)	87.5 (spaces)*

   When using a 1" left/right margin, the margins are:

10 pitch	12 pitch	Proportional
10	12	11**

   Subtract pmargins from length of paper.

85	102	87.5
-20	-24	-22
65	78	65.5

   The results are the total columns (spaces) you can use for the text and the spaces between columns.
4. Subtract results of #2 from results of #3.

65	78	65.5
-46	-46	-46
19	32	19.5

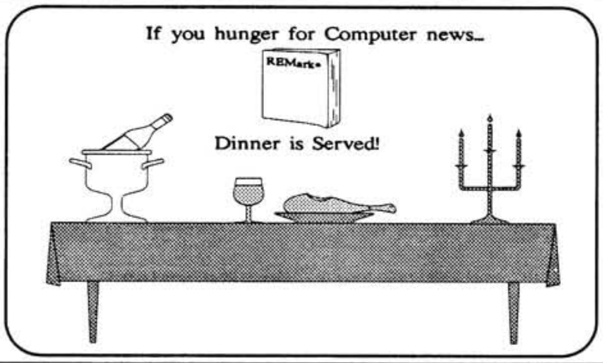
   The results are the total columns (spaces) available for spaces between columns.
5. Divide results of #4 by the # of columns minus one.
   3 columns - 1 = 2

19/2 = 9	32/2 = 16	20/2 = 10

   The results are the spaces between columns.

* The width of paper in proportional space is determined by the typestyle used. To determine the width, check the Paper Size (Shift-F8, p(age)). Round off all calculations.
** The left/right margins are determined by the typestyle being used. To determine the margins, check l/r margins (Shift-F8, l(ine)).

Tab Math Table



Quality Enhancements!

EaZy PC Products

EZM-128: Expand 512K base memory to 640K. Simple, plug-in installation. $95.00
EZCLOCK: Calendar/Clock. Piggy-back add-on for EZM-128. $35.00

No Slot Clock/Calendar

FBE SmartWatch: Automatic date/time on bootup. Installs under BIOS/Monitor ROM. Ten year battery. Works with all Heath/Zenith MSDOS computers. For PC's $32.00, Z-100 $33.00 Module: $25.00

H/Z-148 Expansions

ZEX-148: Adds one full-size and one half-size expansion card slot. $79.95
ZP-148: PAL chip expands existing 640K memory to 704K. CGA/MDA only! $19.95

Configuration Control

CONFIG MASTER: Menu-select active CONFIG.SYS during bootup. Software for PC/Z-100 MSDOS. $29.95

H/Z-150 Items (Not for '157, '158, '159)

VCE-150: Eliminate video card. Install EGA or VGA card. All plug in. Includes circuit board, SRAM and RM-150. $49.95
ZP640 PLUS: PAL chip to expand standard memory card to 640/704K with 2 banks of 256K RAM chips (not included). $19.95
ULTRA-PAL: Three PAL chips: MR150 for 704K + 512K RAM Disk; MR150T for 640K + 512K RAM Disk; LIM150 for 640K + 512K (32 pages) of simulated v3.2 Lotus/Intel/Microsoft Expanded Memory. With software. Install on standard memory card. No soldering. Needs 45 256K RAM chips (not included) for maximum memory configuration. $39.95
COM3: Change existing COM2 to COM3. Put internal MODEM at COM2. Don't lose serial port. With software. $29.95

H/Z-100 Modifications

ZMF100A: Expand "old" motherboard (p/n 181-4917 or less) using 256K RAM chips (not included). No soldering. $65.00
ZRAM-205: Put 256K RAM chips on your Z-205 board. Get 256K plus 768K RAM disk. Contact us for data sheet before ordering. Without RAM chips. $39.00

H/Z-89 Add-Ons

H89PIP: Parallel printer 2 port interface card. With software. $50.00 Cable $24.00
SLOT4: Add fourth expansion slot to right-side accessory bus. $39.95

Order by mail, FAX, telephone, or see your dealer. UPS/APO/FPO shipping included. VISA/MasterCard. WA residents add 8.1% tax. Hours: M-F 9-5 PST. We return all calls left on our answering machine!

FBE

FBE Research Company, Inc.
P.O. Box 68234, Seattle, WA 98168
206-246-9815 Voice/FAX TouchTone Selectable

QUIKDATA - 15 YEARS OF H/Z SUPPORT!

YOUR H/Z ENHANCEMENT EXPERTS!

ACCELERATE YOUR PC/XT/AT!

From Sota Technologies, Inc., the fastest and most proven way to give new life to your H/Z PC/XT computer, giving it -AT compatible speeds! Turn your turtle into a -286 rabbit with a 12.5 Mhz 80286 or 80386 16Mhz SX accelerator board. Complete with 16K on-board CACHE.

The EXP12 **286i** is the effective solution, making your H/Z150/160/150/158/159 series of computers, or any general PC/XT computer faster, in many cases, than a standard IBM AT type computer! **You won't believe your stop watch!**
EXP-12 - **$275**
EXP386 - **$449** Much faster 16Mhz 80386 SX version

For your H/Z241 or 248 we have the MicroWay 20Mhz 386SX accelerator which plugs into a slot and cables to the CPU. Run enhanced mode software such as Windows V3. 32K Cache with optional 32K add-on for 64K cache. Landmark 27, Norton SI 22.4. Cable extra, specify which is needed.
MWFCACHE - $450 20Mhz 386SX Accelerator card
MWCLCC - $95 LCC interconnect cable
MWCPLCC - $95 PLCC interconnect cable
MWCA32 - $65 32K Cache add-on option
387SX20 - $149 Optional 20Mhz coprocessor

MEMORY UPGRADES

Note: All memory upgrades come without memory chips. 150ns 256K DRAMs are $1.79 as of this printing.

Z150MP - $17 Will allow you to **upgrade your H/Z150/160 to up to 704K** on the main memory board, using up to 18 256K DRAM chips.

EAZYRAM - $89 Upgrades EaZy PC from 512 to 640K

ZMF100 - $55 Will allow you to **upgrade your H/Z110/120** (old motherboards; with p/n less than 181-4918) **to 768K system RAM**. Requires 27 256K DRAM chips.
Z100MP - $55 Similar to ZMF100 above, but for new motherboards with p/n 181-4918 or greater.

3MB RAM BOARD for Z241/248 computers is an excellent memory card. Will backfill your 512 to 640K, and provide both extended and expanded RAM; all can coexist. Uses 100ns M256-10 RAM chips, 36 per megabyte desired. Minimum of 18 DRAM chips required ($1.95 each).
EVATRD - $119

Z248/12, Z286LP RAM UPGRADE Z605-1 consists of 2MB SIMM 80ns RAM kits to upgrade your H/Z systems.
Z605-1 - $149

Z386/20, Z386/25, Z386/33, Z386 EISA 2MB SIMM 80ns UPGRADE to add increments of 2MB to these systems. Two required.
ZA3600ME - $89

ZA3800MK - $349 4 megabyte SIMM upgrade for Z386/20, /25, /33, /33E. Must have 4-1 meg SIMMs installed first.

WINCHESTER UPGRADE KITS

PCW20 - $239 Complete MFM winchester setup for a H/Z150, 148, 158, 159, 160, PC etc. Includes **21 meg** formatted half-height Segate ST-124 37ms drive, controller, cable set, doc.
ST-124 - $195 Bare drive only

PCW30 - $295 32 meg with 28ms Segate ST-138-1.
ST-138 - $249 Bare drive only
PCW40 - $319 42 meg with 28ms Segate ST-251-1.
ST-251-1 - $269 Bare drive only
PCW80 - $595 80 meg with Segate ST-4096 full size drive.
ST-4096 - $549 Bare drive only

We also have the DTC controllers ($59) and daughter board expansions ($65) to place a hard drive in the **H/Z148** computers.

WDCON - $59 PC/XT hard drive controller board
WDATCONF - $95 1:1 interleave HD/floppy controller for AT's

FLOPPY DRIVE SAMPLE

MF501 - $71 5" 360K DS/DD drive
MF504 - $79 96 TPI 1.2 meg AT/Z100 drive
MF353 - $79 720K 3.5" drive in 5" frame
MF355 - $85 1.4 meg 3.5" AT drive in 5" frame
TM100-2R - $69 40tk DS reburb (H8/89/Z100 PC type)
Also other drives and full line for older systems

ADD AN EXTERNAL HARD DRIVE TO ANY LAPTOP OR PC DESKTOP with our EXPORT hard drive. Plugs into a parallel port (do not lose the port) to give you an affordable way to easily add a hard drive to your computer. Fast!
EWIN20 - $495 20Mb unit **EWIN40** - $625 40Mb unit

ANY DRIVE IN YOUR PC/XT/AT

With the CompatiCard, you can install up to four additional drives, of any type in your PC/XT/AT computer. Add a 1.2 meg 5" floppy, or a 1.44 meg 3.5" floppy, or any other drive, including 8" to your system. The CompatiCard (CCARD) will handle up to four drives, and the CompatiCard II (CCARD2) will handle up to 2 drives. CCARD4 has boot ROM to allow it to be used as primary boot controller in systems that allow you to remove floppy controller in some systems. Also handles 2.8MB 3.5" floppy drives. Additional cables and external enclosures may be required.
CCARD2 - $79 **CCARD** - $99 **CCARD4** - $125

ADD ANY FLOPPY TO ANY LAPTOP OR PC WITH A PARALLEL PORT easily and inexpensively. With Backpack, you simply connect the external unit to your parallel printer port (do not lose printer function), install software, and away you go! No expansion boxes needed for laptops, and no slots required. Too easy to be true, but it is. Want a 2.8mb/1.4mb/720K floppy on your MinisPort? Want to add a 1.2 meg to your laptop? Want to add an additional drive to your desktop? Plug it in and go. 2.8MB 3.5" version will read and write 2.8 meg, 1.4 meg, and 720K format. 1.2 meg version will read 1.2 meg and 360K and write 1.2 meg format.
BPACK2.8 - $319 **BPACK1.4** - $269
BPACK1.2 - $269 **BPACK360** - $269

8-BIT/Z100

We carry a full line of replacement boards, parts and power supplies for the H/Z89/90 and Z100. We also have some H8 boards available. We continue to fully support and carry a full line of hardware and software products for the H8/H89/90 and Z100 computers including almost the full line of Software Toolworks items. Other items we carry include diskettes of all types, printers, modems, hard drives, etc.

OTHER STUFF

Quikdata also carries BIOS ROM upgrades and batteries for most H/Z PC/XT/AT computers, spike protection filters, backup power supplies, tape backup units, modems, printers, cables and ribbons, disk drives and diskettes of all types, external hard drive and floppy drive enclosures, cables and connectors, laptop batteries, CMOS batteries, video monitors and video cards, memory cards, memory chips and ICs, joysticks, accessory cards, serial and bus mouse, a variety of useful and most popular software and much more! **Need a PC/XT/AT computer?** Tell us what you want and we will quote you a price on one of our custom assembled QD computers made up to your exact specifications!

Call or write in to place your order, inquire about any products, or request our free no obligation catalog. VISA and Master Card accepted, pick up 2% S&H. We also ship UPS COD and accept purchase orders to rated firms (add 5% to all items for POs). All orders under $100 add $4 S&H. Phone hours: 9AM-4:30PM Mon-Thu, 9AM-3PM Friday. Visit our bulletin board: (414) 452-4345. FAX: (414) 452-4344.

QUIKDATA, INC.
2618 PENN CIRCLE
SHEBOYGAN, WI 53081-4250
(414) 452-4172

Adding a 2.88MB 3.5" Floppy Drive to Your Computer

Ron Siebers
9334 Wentlock Road
Woodbury, MN 55125-3420

How would you like to be able to store the contents of eight 5-1/4", 360K floppies on one disk. That is almost 3 megabytes of file space that you can slide into your pocket. Interested? Read on!

New Extra High-Density (2.88 MB) Floppy Drives

Many still think you can't use high-density 3.5" diskettes on an 8-bit computer, such as the IBM PC-XT or their clones. This is not true. A number of third party vendors offer hardware that connect a 3.5" high-density 1.44 Mb drive to any IBM or compatible computer. One such vendor whose products I have used for several years is MicroSolutions. Their newest products are the CompatiCard IV, Megamate and Backpack systems. Each of these supports a new extra high-density 2.88 MB 3.5" floppy drive. The drive uses a new 4 M-byte raw capacity 3.5" diskette that formats to more than 2.88 M-byte of storage. Toshiba pioneered the barium ferrite medium for these new diskettes that are now licensed for manufacture by 3M. The disks are expensive right now, from over $10 per diskette suggested list to under $8 per disk at some dealers. Prices should drop as usage increases. If you are considering adding a 3.5" drive to your PC, I strongly recommend going with this new 2.88 M-byte drive over a 1.44 M-byte drive to provide for the future. Even if you don't use the 2.88 M-byte disks now because of the high cost, you will be able to in the future when costs come down. Other reasons to buy an extra high capacity drive are:

- It offers twice the storage capacity on the same physical size diskette, allowing larger file sizes to be backed up from hard drives.
- It offers 1 M-byte/second data transfer rate, twice as fast as 1.44 M-byte drives.
- It will still read, write and format the 720K and 1.44 M-byte diskettes without any problems.
- IBM introduced a new computer this past June with a 2.88 MB 3.5" floppy drive, creating a new drive standard for PC's you buy in the future.
- The just released DOS 5.0 will support formatting for this new standard.
- The additional cost for the capability of doubled storage capacity is only about $45 more than the 1.44 Megabyte version.

CompatiCard, Megamate or Backpack?

Microsolutions offers three ways to add drives to your computer. The CompatiCard is an 8-bit short slot card that fits in any 8-bit or 16-bit ISA or EISA buss slot (but not MCA or Microchannel slot computers). It can replace the original dual or four floppy controller used on any IBM PC, XT or their clones. In these computers you won't lose a slot. The IBM AT, and most clone 286 and 386 computers including Zenith Data Systems have a dual hard/dual floppy controller card, so you must have an unused slot available for the CompatiCard. The CompatiCard IV can connect up to four floppy drives of any size or density to a computer. These can be four internal drives or two internally mounted and two external drives. The products provide the following upgrade capabilities for older PC's:

- Add one or two 3.5" high-density external drives to a dual floppy IBM PC or XT without a hard drive to provide larger floppy disk storage to run larger programs and provide compatibility with 3.5" drives in newer computers.
- Add one or two 3.5" high-density external drives to an IBM PC, XT, AT or compatible to provide larger backup capability for hard drive files and compatibility with 3.5" drives in newer computers.
- Add one to four more floppy drives to any 286 or 386 computer that only allows two floppy drives.
- Add an extra high-density 2.88 MB drive to any of the preceding scenarios.

The CompatiCard has address jumpers that allow it to be configured as the only drive controller or as a 2nd, 3rd or 4th drive controller. DMA channel and BIOS address selection are also provided. Version IV of the CompatiCard adds two important capabilities to previous versions. The first is support for the new Extra High-Density (2.88 MB) floppy drives. The second is the ability to check any of the four drives you can connect to the controller for a bootable disk and boot from it. The card can supply its own ROM BIOS on bootup and a jumper activates the floppy drive boot check.

In PC's and XT's, I use the CompatiCard to replace the original controller, connecting the original two internal drives as before and an added external 3.5" drive via the rear connector. You can order the CompatiCard alone if you want to add drives you already have or those you can buy cheaper separately. It comes with a software driver that is called from a line added to your CONFIG.SYS file. It also includes a formatting program you use in place of the DOS FORMAT.COM and a utility program that allows you to use a pop-

Table 1
Floppy Disk Sizes/Densities

Drive Size	Unformatted Size/3M Disk Label	Formats To	3M Disk Color
5.25"	500 KB Double-Sided, Double-Density	360K	Black
	1.6MB Double-Sided, High-Density	1.2 MB	Black
3.5"	1 MB Double-Sided, Double-Density	720 KB	Lt. Gray
	2 MB Double-Sided, High-Density	1.44 MB	Black
	4 MB Double-Sided, Extra High-Density	2.88 MB	Dk. Gray

up format menu and to format in the background while you use another program. Many programs in their current versions can no longer fit on the 360K floppies and are therefore unusable on dual floppy PC's that do not have a hard drive. Adding a hard drive, such as a Plus Systems Hardcard, is more expensive than adding a CompatiCard and a 3.5" drive. A high-density and certainly the extra high-density drive would provide enough storage on one floppy to run some programs that normally require a hard drive from floppies. In an AT or older 286 clone, CompatiCard can allow the addition of a 3.5" drive, either internally as a second floppy drive or externally as a third drive.

One important note about adding 3.5" drives to IBM computers. You should obtain a device driver called DASDDRVR.SYS from IBM, copy it into the root directory and call it from a "DEVICE=" line in CONFIG.SYS. This is necessary to assure compatibility between 3.5" disks formatted in drives added to older IBM computers and those in newer clone computers. Failing to do this will result in 3.5" disks formatted in IBM computers to sometimes give a "drive read failure" followed by the dreaded "Abort, Retry, Ignore" message in a clone. This frequently happens to those using an IBM at work and taking work home to use on their Zenith Data Systems computer. Reformatting the 3.5" disks with this driver in effect corrects whatever clone compatibility problem exists.

The Megamate system consists of a CompatiCard IV and your choice of floppy drive in an external cabinet. You can order a Megamate with any of the drive sizes and densities shown in Table 1. The drive comes in a very small cabinet with a short cable that attaches to the 37-pin connector on the end of the CompatiCard IV. The drive gets its power from the computer card buss through the cable and CompatiCard. Although the drive only draws 4 watts, you may need to upgrade to a larger power supply in a PC or XT that still uses the original 65 Watt power supply, if you install the Megamate system. You have to add the current draw of the CompatiCard also. Most clones like Zenith Data Systems have a larger power supply. If it is 100 watts or higher, you probably don't need more.

Backpack is Microsolutions' external drive system that connects to the parallel printer port on any IBM PC or compatible. Relax, you can still connect and use your printer on the same port via a parallel connector on the rear of the Backpack. The benefit of this system is that you don't need a slot for the CompatiCard. The controller electronics and power supply for the drive are built into the drive cabinet and provide external operation, but only for one drive. Another benefit of the Backpack is you can use it as a shared backup drive. Users can easily disconnect it and take it to other computers for hard drive backup on floppies. Again, your choice of drive size and density are the same as discussed for the Megamate.

Table 2
MicroSolutions Drive Pricing

Floppy Drive Product	MicroS List	Quikdata
CompatiCard IV (controller only)	$149.00	$125.00
3.5" 2.88 MB drive in 5¼" mount	$199.00	$159.00
Megamate (1.44 MB)	$295.00	$250.00
Megamate 2.88 MB	$395.00	$295.00
backpack 360K, 1.2 MB or 1.44 MB	$349.00	$269.00
backpack 2.88 MB	$425.00	$305.00
2.88 MB Toshiba Floppy Disk (ea)	$ 9.95	$ 7.40

Installation

Installing either the CompatiCard or the Megamate requires the following sequence based on my experiences:

1. Read manual and set CompatiCard jumpers. The default jumper settings are for controller 1, the only hard drive or floppy controller card in the computer. If the CompatiCard is to be the second, you need to change some jumpers.
2. Remove the computer cover.
3. Remove the blank slot plate and install the CompatiCard.
4. Install the added internal floppy (if used). See the floppy manual for any jumpers. Remember that the drive on the end of the cable (after the twisted wires) is the first drive. If you connect a second drive to the middle connector, DOS assigns it as a second drive on that cable. Be sure the cable stripe is on the left.
5. Connect internal drives (if used). On computers that have a hard/floppy drive controller, I leave the internal drives connected. This maintains the drive A: and B: designations you are probably used to. If you transfer them to the CompatiCard, it is extra work and DOS assigns the next drive ID after your hard drive designation (usually D: & E:).
6. Reinstall the computer cover.
7. Connect Megamate or other external drive to the connector on the CompatiCard bracket.
8. Run install (some versions) or copy the software to a COMPCARD or UTILITY directory on your hard drive.
9. Add the statement "Device=MMDRIVER.SYS" to CONFIG.SYS or whatever driver your chosen product provides.
10. Reboot the computer. You should be able to select the new drive(s) and format a disk.

MicroSolutions provides software, some of which is mandatory and some of which is optional. The device driver name varies, depending on which of the three products you choose. The Megamate uses MMDRIVER.SYS. You must copy this file to the root directory or a subdirectory and call it in CONFIG.SYS. MMFORMAT must be used to format disks in any of the MicroSolutions' drives. You still use DOS' format for drives controlled by the original floppy controller. Since the computer hardware and, therefore, DOS do not know about the added drives, format will give an error message if you try to use it on an added drive. BACKFMT.COM is an optional TSR program that allows you to format floppies in the background while you use other programs.

More information on these products is available from:

The Manufacturer	*A Dealer*
MicroSolutions	Quikdata Inc.
132 W. Lincoln	Highway2618 Penn Circle
DeKalb, IL 60115	Sheboygan, WI 53082-4250
(815) 756-3411	(414) 452-4172

MicroSolutions' format program for the added drives works a bit different than DOS's Format. The manual describes the command options. As is normal for high-density drives, if you just say MMFORMAT E:, the format program assumes that you are formatting a disk that matches the drive density. MMFORMAT assumes you have a 2.88 MB disk in the Extra High-Density drive. To format a 1.44 MB disk in this drive, you must specify the density by number. The correct command is "MMFORMAT E:/1.4". The usual DOS switches still apply, /S for system files, /V for volume label, etc. To format a 720K disk with a volume label, use "MMFORMAT E:/720/V".

Pricing

If all of this has been of interest to you, you probably want to know "how much?". See Table 2.

Continued from Page 10

memory that can be individually switched into a portion of the PC's memory map and, thus, be made available to the host CPU. For example, when 256K of memory is installed, there are four separate 64K banks. To implement a 16-color EGA mode, 4 bits are necessary to identify a desired color. This pixel-plane method, as it is sometimes called, stores each of the four required bits in the four separate planes, with one bit in each of the four planes.

The actual programming of the EGA, especially for the graphics modes, is much more complex than either the MDA or CGA. Entire books have be devoted to explain this subject, so I will not attempt to try here.

Video Graphics Array (VGA)

When IBM introduced its line of PS/2 MicroChannel Architecture computers in 1987, they also introduced a new video subsystem called the Video Graphics Array, or VGA. The architecture of the VGA is very similar to the previous EGA with three major differences: new modes were added to support higher resolution and color depth, there was a switch from digital signal output to analog signal output, and 256Kb of video memory is installed.

The VGA's maximum resolution was increased from 640x350 in the EGA to 640x480. This increase in resolution not only allowed for greater graphics resolution, it also improved the readability of standard text operations. In the VGA, the character block size was increased to 9 x 16 pixels, up from the EGA's 8 x 14.

On the graphics side, three new modes were added: 11H, 12H, and 13H. By far the most popular mode utilized is 12H, a 640 x 480 resolution with 16 simultaneous colors allowed.

Like the EGA, the VGA is also a complex device and is not a trivial task to program. For those inclined to VGA programming, the subject is addressed by many books in your local bookstore.

SuperVGA

Within a year after the VGA was introduced, various video board manufacturers were able to duplicate IBM VGA functionality in their products and soon a flood of VGA clones hit the market. Many of these video vendors began to add extra features and additional operating modes to help differentiate their products from their competitors. The most popular was to add a capability for greater resolution and color depth.

The problem with having a multitude of vendors producing SuperVGA video boards was that these boards tended to be proprietary in nature, and almost always required special software drivers for a particular software application (i.e., Windows, AutoCAD) to work correctly with the video hardware. In the absence of IBM providing a lead for everyone else to follow, the various video equipment vendors finally realized the need to band together in an attempt to standardize technical aspects of SuperVGA. This group is known as the Video Electronics Standards Association (VESA).

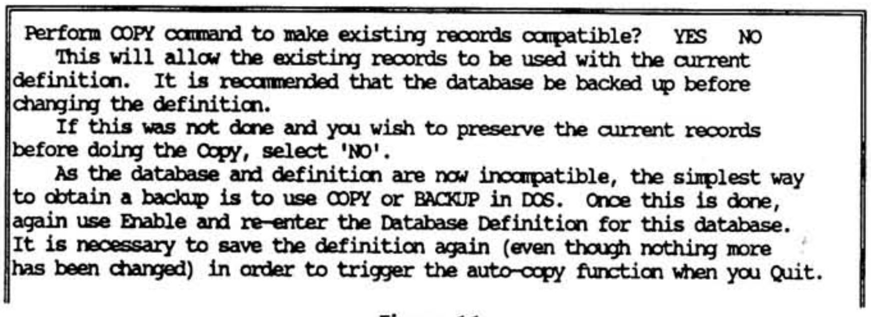
Other IBM Video Adapter Types

Outside of the above mentioned video adapters, several others have reached the marketplace, some have been mildly successful, others have been total flops. IBM has been the primary contributor to PC video solutions (no surprise here), but they have also introduced other video solutions that are not really considered "standards". This includes the 8514/A, the Memory Controller Gate Array (MCGA), a Professional Graphics Card (PGC), and the Extended Graphics Array (XGA).

34010/TIGA

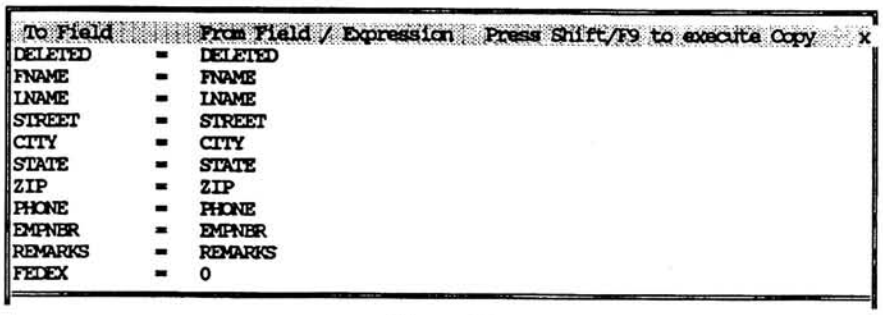
One type of video solution that has remained independent of IBM's way are adapters based on Texas Instruments' 34010 or 34020 line of graphics-oriented processors. These adapters are specialized co-processor cards that operate independently of the primary video solution (i.e., VGA), the system BIOS, and the operating system. Their specialty is almost always high-resolution graphics, higher color depth, and overall faster graphics operation. Some of the manufacturers of 340x0-based video cards utilize proprietary software to run (GEM and DGIS), but those that utilize the Texas Instruments Graphics Architecture (TIGA) software interface standard are gaining popularity very quickly. TIGA's major advantage is that application software requirements allowing a particular program to communicate with the video hardware are greatly reduced compared to all other non-PC-compatible video solutions. Zenith Data Systems has determined that a TIGA-based 34010 board is the best solution for all of its video requirements beyond VGA.

Continued from Page 20



```
 Perform COPY command to make existing records compatible?   YES   NO
    This will allow the existing records to be used with the current
definition.  It is recommended that the database be backed up before
changing the definition.
    If this was not done and you wish to preserve the current records
before doing the Copy, select 'NO'.
    As the database and definition are now incompatible, the simplest way
to obtain a backup is to use COPY or BACKUP in DOS.  Once this is done,
again use Enable and re-enter the Database Definition for this database.
It is necessary to save the definition again (even though nothing more
has been changed) in order to trigger the auto-copy function when you Quit.
```

Figure 11

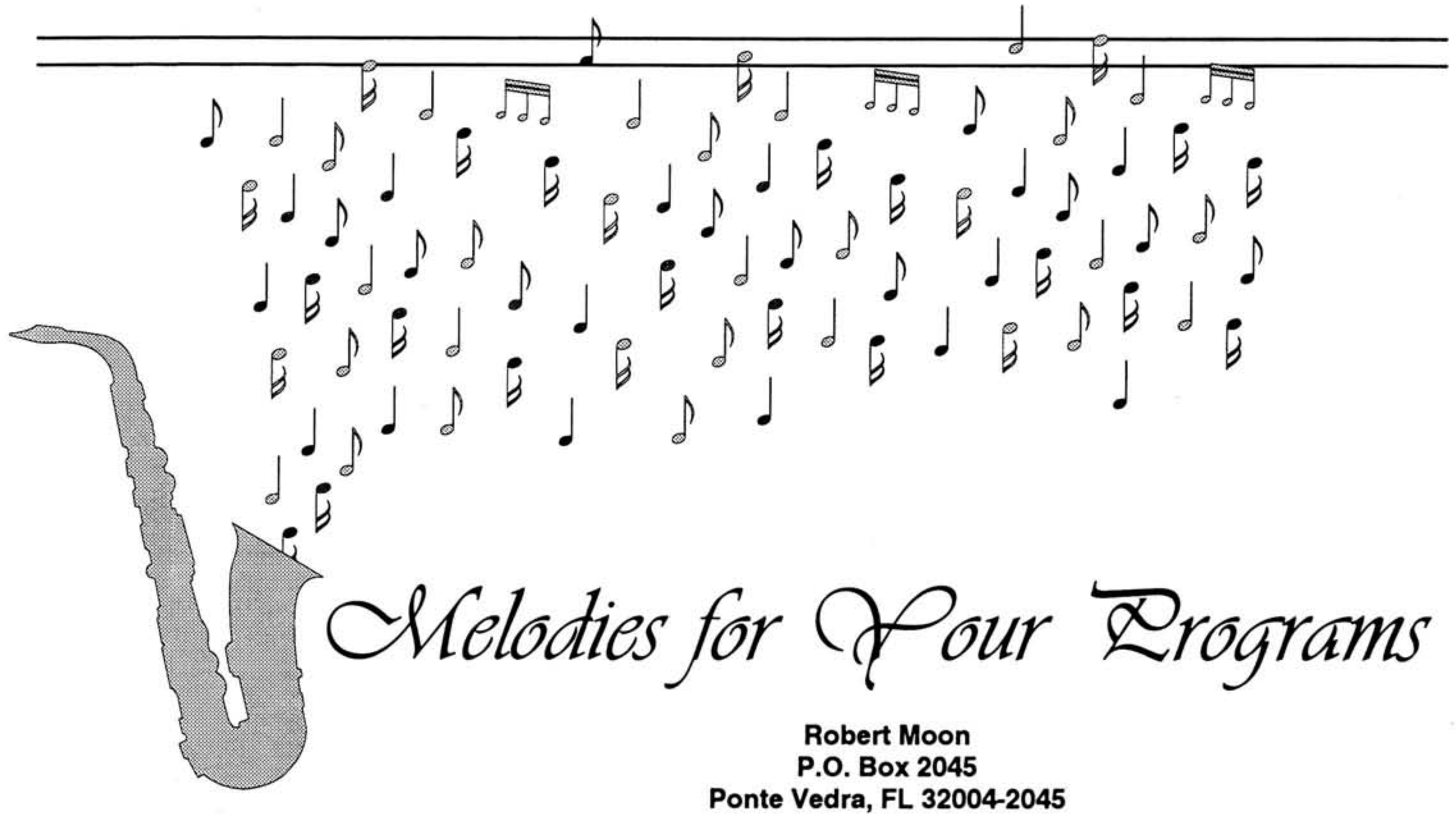


```
To Field          From Field / Expression   Press Shift/F9 to execute Copy   X
DELETED     =     DELETED
FNAME       =     FNAME
LNAME       =     LNAME
STREET      =     STREET
CITY        =     CITY
STATE       =     STATE
ZIP         =     ZIP
PHONE       =     PHONE
EMPNBR      =     EMPNBR
REMARKS     =     REMARKS
FEDEX       =     0
```

Figure 12

Classified Ads

ZENITH Z-100 COMPUTERS, dual floppy drives, used, complete, in operational condition. 15 to sell $150.00 each plus shipping. Call Frank Esposito (407) 657-2287.



Melodies for Your Programs

Robert Moon
P.O. Box 2045
Ponte Vedra, FL 32004-2045

I use the bell or beeps from the PC's speaker often in my programs. They signal completion of a process, an error, termination of a program, prompt for input, or just to get attention.

I like to put 'bells and whistles' in my programs, so the first time that I heard a musical tune from a PC, it occurred to me that a melody at appropriate times in a program would be more pleasing than the normal beeps.

There is no magic to it. In fact, it is quite simple. All the hardware is built in – the PIT (programmable interval timer) chip 8253 (8254 on the AT), the PPI (programmable peripheral interface) chip 8255 (MC146818,RT/CMOS on the AT), and of course, the speaker. This is all the hardware needed to make sounds or produce a tune.

The PPI contains three 8-bit registers, A, B, and C. Each register is connected to the CPU (central processing unit – 8088, 80286, 80386, etc.) through a port. These ports are labeled, using hexadecimal, 60h, 61h, and 62h, respectively. Ports 60h and 62h are read only. Port 61h is read/write and the one we're going to use.

Here's the function of each bit of port 61h:

Bit	**Function definition**
0	Timer 2 gate
1	Speaker data
2	Read RAM size or read spare key
3	Cassette motor off
4	Enable RAM
5	Enable I/O channel check
6	Enable kbd clock signal
7	Enable kbd or enable sense sw's

Bits 0 and 1 control the speaker, so they are the bits of interest. Sending a 3 to port 61h will turn the speaker on. Sending a 0 will turn it off. Since this port controls other devices as well, the safe way is to get the current port settings, OR it with a 3 to turn the speaker on, then restore the original settings to turn the speaker off.

The PIT contains three independent 16-bit counters. Counter 1 controls the system clock, counter 2 controls the speaker, and counter 3 controls memory refresh. The output of counter 2 is ANDed with the speaker data from bit 1 of port 61h. Bit 0 of port 61h provides a gate for counter 2, which enables counting. There are also ports associated with the PIT. Port 42h is read/write that allows loading and reading the contents of counter 2. Port 43h accesses the control word, an 8-bit register, which is write only. This control word register accepts information for programming the operation of each counter. Figure 1 shows the control word format.

Now we have to program the PIT so that it is set up in the correct state for producing a tune.

The BCD bit chooses either a 16-bit binary counter or a binary coded decimal counter. We want the binary counter, so a 0 will go into BCD. There are six modes in which the counter can operate. We want the counter to generate square waves, which is mode three. Therefore, a 1 will go into M0 and M1, and a 0 will go into M2. We will send the frequency information in two bytes. The counter will have to read two bytes. To do this, a 1 is put into both RL0 and RL1. The least significant byte is sent first. We will use counter 2, so a 0 is needed in SC0 and a 1 in SC1. After setting the appropriate bits, our control word will be 10110110 binary or 0b6 hex. This byte will be sent to port 43h.

The frequency of the square wave is determined by the count down value, the divisor. The counter divides this divisor value into the PIT clock frequency of 1193182 Hz to produce the desired frequency. We get the divisor by dividing the clock frequency by the desired frequency. There is a reciprocal relationship between the count down value (the divisor) and the frequency. We then send this two byte divisor to port 42h. The low order byte is sent first, then the high order byte. The speaker is then turned on and pulsed at this frequency. The sound generated will continue until the speaker is turned off.

Bit		**Function Definition**
0	(BCD)	Binary coded decimal, yes or no
1,2,3	(M0,M1,M2)	Counter mode of operation
4,5	(RL0,RL1)	Read/load specification
6,7	(SC0,SC1)	Select counter

Figure 1

We must have some means to time the speaker's on period and then turn it off after the specified duration. We will use the time of day clock to set up a delay loop. First, we'll get the current time of day using DOS int 21h, function 2ch. Then we'll add our delay values to the current time. This will be the time at which the speaker should be turned off. Now, we'll just keep checking the current time until the time is up. And that's when we turn the speaker off.

Well, we know how to manipulate the hardware to get the PC to make sounds. Now we must learn something about music so that we can get it to play a melody.

A melody is a succession of tones that make up a pleasing tune. Melodies are constructed from the eight tones on a scale. A scale is a pattern of ascending or descending tones within an octave. These are natural tones and are designated by the first seven letters of the alphabet and begins over again at every eighth tone. This is because the eighth tone, or octave, is just twice the frequency of the original tone. Going up one octave precisely doubles the frequency. Going down one octave precisely halves the frequency. Each group of seven steps plus the octave, or 8th step, is built on a uniform model of ratios. The ratio of the frequency of A to B, C to D, and F to G is roughly equal and is considered to be a whole step. The frequency ratio of E to F and B to C is only half as large and, therefore, is considered to be a half step.

To make a scale of half-steps or semitones, the whole step tones are sharpened or flattened, producing five additional tones called accidentals. The symbol for a sharp is similar to the number or pound sign (#). The symbol for a flat is similar to a lower case b, see Figure 2. The symbols follow the letter, when spoken or written, as F# (F sharp) or Gb(G flat), but will precede the note on the staff. The black keys of a piano produces the accidentals, the white keys the naturals.

The result is twelve semi-tones called

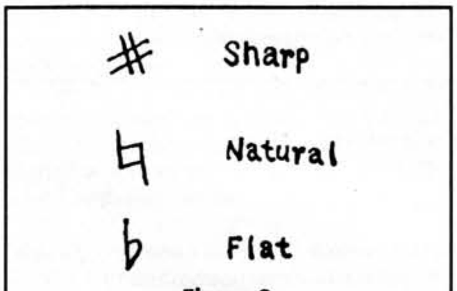


Figure 2

an equal tempered chromatic scale.

The characteristics of sound are pitch, loudness, timbre, and duration.

Pitch is the regular or even frequency of sound. The piano has a frequency range from 27Hz to 4186Hz, spanning over seven octaves. Loudness is the intensity of sound. It depends on the amplitude of the tone.

Timbre refers to the harmonic content or overtones. Overtones make it possible to identify different instruments. It's the basis of their quality.

Duration is the length of time that a tone sounds. Tones can last from fractions of a second to many seconds.

To keep the program smaller and simpler, we will control only the pitch and duration. Music is written using a system of notation that has evolved through centuries of invention and experimentation. There is a certain group of fundamental symbols now in use that make up our present day system. These symbols are notes, staff, and clefs.

A note, see Figure 3, is a written symbol used to indicate the pitch and duration of a musical tone.

The staff, see Figure 4, is a group of five

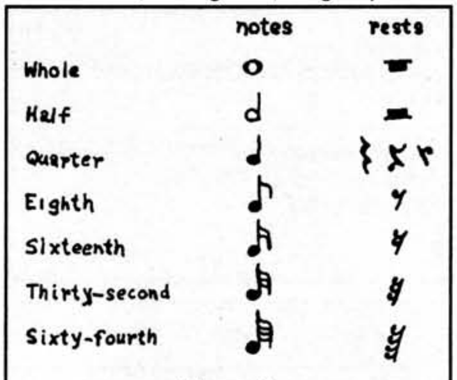


Figure 3

parallel lines enclosing four spaces. The staff is cut into segments by vertical lines, called bars. The space between bars is called a measure. The end of a piece of music is marked by a double bar.

The clef is a sign at the beginning of the

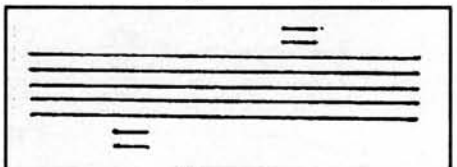


Figure 4

staff. Clefs help us by giving the position of one note. This enables us to work out the other notes.

The most common is the G or treble clef, see Figure 5, which gives the position of the note G (in the one-lined octave) on the second line of the staff. Another clef often seen is the F or bass clef, see Figure 6, showing the position of F (in the small octave) on the fourth line. The lines are counted from the bottom.

Pitch is shown by the position of the note on a line or space. When it becomes necessary to include pitches that are above or below the staff, ledger lines can be

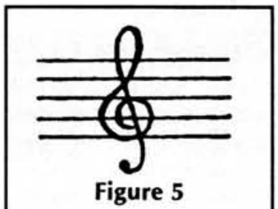


Figure 5

added. In Figure 4, the ledger lines are the two short lines shown above and below the staff.

Duration is indicated by the shape of the note. This representation is relative to the speed or tempo

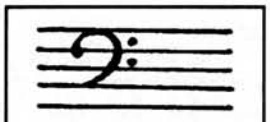


Figure 6

that the melody is played. A dot following the note extends the duration by one half. Two dots will extend it by three quarters.

There can be periods of silence in music. These periods are known as Rests, see Figure 3. Rests are counted the same as the notes they replace.

Music has different levels of loudness or volume. But, unfortunately, we can't vary the volume of the PC's speaker. However, I did have to attentuate the signal going to the speaker of my computer. It was causing distortion by being overdriven. A 10k potentiometer placed between the output connection and the speaker permitted adjustment to eliminate the distortion. This pot could be replaced by fixed resistors.

Sharps and flats may be written into music as they are needed. To save unnecessary writing, the essential sharps or flats may be placed at the beginning of a piece of music immediately after the clef sign on the proper line or space, see Figure 7.

This arrangement of the accidentals at the beginning of a piece is called the key signature. An accidental in the key signature affects all notes of that letter-name, at all octaves, thoughout the piece, unless changed by another accidental or natural symbol.

The meter, or time signature, see Figure 7, follows the clef on the staff. It is shown as two numbers, written one above the other, such as 4/4, spoken 'four four'. The upper number indicates the number of notes and the lower number the type of note.

In 4/4 time, there are four quarter note

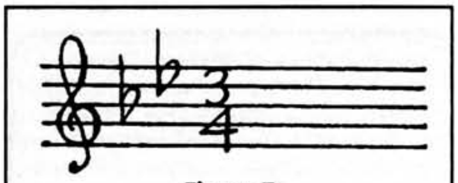


Figure 7

beats to a measure or bar. There are three beats to a bar in 3/4 (three four) time, two beats in 2/4 (two four) time. 4/4 is common time (sometimes shown as a large letter C), 3/4 is waltz time, and 2/4 is march time.

The meter signature refers to the number of beats to a bar. In the case of four four time, this can be four quarter notes, two half notes, or one whole note.

Music can be played fast or slow. The speed or tempo at which a piece of music is played depends upon how the beats, which are regularly occurring pulses, are counted and how the musician feels about it. The composer can, and often does, give an indication of how he intended it to be played.

We will use the number of beats in a minute to indicate tempo. There are four beats in a whole note, two in a half note, one in a quarter note, one-half in an eighth note, one-fourth in a sixteenth note, one-eighth in a thirty-second note, and one-sixteenth in a sixty-fourth note.

This has been an extremely brief introduction to music notation, but should enable you to translate a simple music score into the data format required by the program. See the melody format in Figure 8.

Three procedures make up the program in Listing 1: PLAY, TONE, and DELAY. Your program will point to the desired tune, then call PLAY. PLAY fetches bytes from the note table and processes the data to obtain the frequency and duration of the note. PLAY will then call TONE, which will set up the timer for the required frequency, turn on the speaker, and call DELAY. DELAY will add this duration to the time and return when that time expires. Tone will then turn off the speaker and return to PLAY.

In Listing 1, the procedure MAIN is used only to test the three procedures and the melody format. The code there, with the exception of the provision for repetition and exit, would be in your program at the appropriate place to play an appropriate tune. The data would be included in your data segment. The equates would be added to the ones in your program. Only the equates that would be used for your particular melodies need be included. The formatted melody in the table is the old English love song "Greensleeves".

We will use a shorter piece as an example for translating a music score to the program format. Figure 9 is the score for "Swanee River".

We will try a tempo of 120 beats per minute, 'T',Q120. The tempo can be adjusted if it doesn't sound 'right' after playing. The first note is in the L1 octave, so, the entry is 'O',L1. By the shape, we see that it is a half note H and the position indicates an E note; therefore, this entry is H,E. Next is an eighth note D and the entry is I,D. Then we have a ledger line and the note there is an eighth note C – I,C. The remaining two notes in the first measure are also eighth notes – E and D, respectively, I,E,I,D.

The second measure starts on a ledger line. It is a quarter note C. The next note is also a quarter note C, but in a different octave. This is the L2 octave. We will enter that as – 'O',L2 then Q,C. Next is an eighth note A in the L1 octave, the entry is – 'O',L1,I,A. The last note is in the L2 octave, a quarter note C. There is a dot following this note so this entry is written 'O',L2,Q,'*',C. The symbol for a dot immediately follows the duration. To signal the end of the melody we put 'X' as the last entry.

Figure 10 shows the data table for 'Swanee River'.

There was no rest in this short piece, but as an example a quarter rest period is entered as 'R',Q , whenever it occurs, just like a note.

You can have a melody to indicate an error, a different one for completion, and yet another to alert for some response. Perhaps a really great one for an attaboy. Sure beats the plain old 1000Hz tone.

```
'X'     (end of tune),
'T'     (TEMPO—Q48,Q60,Q80,Q120,Q240,Q480,Q720,Q960,Q1200)
'O'     (OCTAVE—CA,GR,SM,L1,L2,L3,L4)
'R'     (REST—W,H,Q,I,S,T,Y,)
        DURATION (W,H,Q,I,S,T,Y, dots are represented as '*')
        NOTE (C,C@,D,D@,E,F,F@,G,G@,A,A@,B;
        the accidentals are indicated by the suffix of @)
```

Figure 8

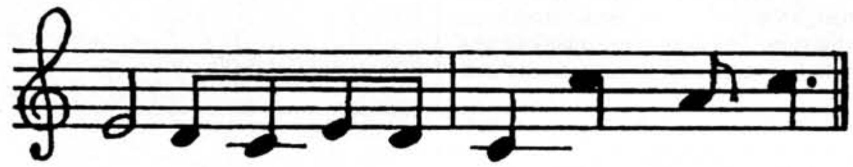
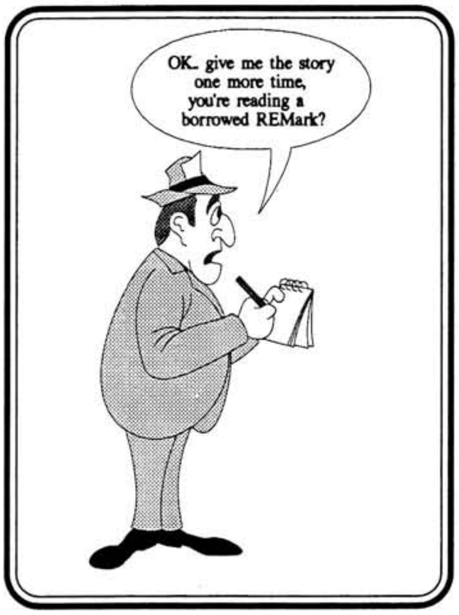


Figure 9

```
SWANEE_NOTE        db      'T',Q120
                   db      'O',L1,H,E,I,D,I,C,I,E,I,D
                   db      Q,C,'O',L2,Q,C,'O',L1,I,A,'O',L2,Q,'*',C
                   db      'X'
```

Figure 10


OK. give me the story one more time, you're reading a borrowed REMark?

Listing 1

```
;           These procedures will produce a melody
;           using the PC's speaker and timer 2
;
;           written by Robert Moon, 17 Jan 91
;
;           copyright (c) 1991 Robert Moon

TITLE  Melody

.MODEL  SMALL

;Equates

;Pitch table. Frequencies are for the bottom octave

C         equ       33          ;C
C@        equ       35          ;C sharp or D flat
D         equ       37          ;D
D@        equ       39          ;D sharp or E flat
E         equ       41          ;E
F         equ       44          ;F
F@        equ       46          ;F sharp or G flat
G         equ       49          ;G
G@        equ       52          ;G sharp or A flat
A         equ       55          ;A
A@        equ       58          ;A sharp or B flat
```

```
B        equ      62            ;B

;Octave range

CA       equu     1                 ;Contra octave
GR       equ      2                 ;Great octave
SM       equ      4                 ;Small octave
L1       equ      8                 ;One-lined octave
L2       equ      16                ;Two-lined octave
L3       equ      32                ;Three-lined octave
L4       equ      64                ;Four-lined octave

;Duration of notes and rests

W        equ      100               ;Whole note
H        equ      50                ;Half note
Q        equ      25                ;Quarter note
I        equ      12                ;Eighth note
S        equ      6                 ;Sixteenth note
T        equ      3                 ;Thirty-second note
Y        equ      1                 ;Sixty-fourth note


;Tempo

Q48      equ      5                 ;48 beats per minute
Q60      equ      4                 ;60 beats per minute
Q80      equ      3                 ;80 beats per minute
Q120     equ      2                 ;120 beats per minute
Q240     equ      1                 ;240 beats per minute
Q480     equ      -2                ;480 beats per minute
Q720     equ      -3                ;720 beats per minute
Q960     equ      -4                ;960 beats per minute
Q1200    equ      -5                ;1200 beats per minute


;Code area

.CODE

;Plays the melody from data in the table


MAIN     proc          near ,

         mov      ax,_DATA ;Initialise ds register
         mov      ds,ax
AGAIN:
         lea      si,GRN_NOTE       ;Point to the tune
         call     PLAY              ;Play it
         mov      dx,offset MESSAGE       ;Ask if want to hear it again
         mov      ah,9              ;Display string (message)
         int      21h
         mov      ah,7              ;Wait for response, any key but n will repeat
         int      21h
         cmp      al,'N'            ;No ?
         jz       EXIT
         cmp      al,'n'
         jz       EXIT              ;No, go to DOS
         jmp      short AGAIN       ;Yes... play it again, Sam
EXIT:
         mov      ax,4c00h          ;Go to DOS
         int      21h

MAIN     endp
```

```
;Data area

.DATA

;Storage

OCTAVE   db  ?                        ;storage for octave range
TEMPO    db  ?                        ;Storage for tempo

;Message

MESSAGE db  'Again ? ','$'            ;Again message

;Duration, octave, and notes table for the tune

GRN_NOTE db     'T',Q80
         db   'O',L1,I,A,'O',L2,Q,C,I,D,I,'*',E,S,F@,I,E,Q,D,'O',L1,I,B,Q,G
         db   S,A,S,B,'O',L2,Q,C,'O',L1,I,A,I,'*',A,S,G@,I,A,Q,B,I,G@,Q,E,I,A
         db   'O',L2,Q,C,I,D,I,'*',E,S,F@,I,E,Q,D,'O',L1,I,B,Q,G,S,A,S,B
         db   'O',L2,I,'*',C,'O',L1,S,B,I,A,I,'*',G@,S,F@,I,G@,Q,A,I,A,Q,'*',A
         db   'O',L2,Q,'*',G,I,G,I,F@,I,E,Q,D,'O',L1,I,B,I,G,I,A,I,B,'O',L2,Q,C
         db   'O',L1,I,A,I,'*',A,S,G@,I,A,Q,B,I,G@,Q,'*',E
         db   'O',L2,O,'*',G,I,G,I,F@,I,E,Q,D,'O',L1,I,B,I,G,I,A,I,B
         db   'O',L2,I,'*',C,'O',L1,S,B,I,A,I,'*',G@,S,F@,I,G@,Q,'*',A,Q,A
         db     'X'


;Stack area

.STACK  100h

;Some more code

.CODE

;Play a tune using data from the table

PLAY          proc near

NOTE:
         cmp     byte ptr[si],'X' ;End of tune?
         jne     $+5              ;No, there's more, so jump over next statement
         jmp     PLAY_EXIT        ;Then quit
         cmp     byte ptr[si],'T' ;Is it tempo?
         jne     NO_TEMPO ;No
         inc     si               ;Yes, point to the tempo
         mov     al,byte ptr[si]  ;Get it
         mov     TEMPO,al         ;And save it
         inc     si               ;Point to next entry
NO_TEMPO:
         cmp     byte ptr[si],'O' ;Is it an octave range?
         jne     NO_OCTAVE        ;No
         inc     si               ;Yes, point to the octave weight
         mov     al,[si]          ;Get the octave
         mov     OCTAVE,al        ;And store it
         inc     si               ;Point to next entry
NO_OCTAVE:
         cmp     byte ptr[si],'R' ;Is it a rest?
         jne     NO_REST          ;No, it's the note duration
         inc     si               ;Yes, point to the rest duration
         mov     cl,[si]          ;Get the rest duration
         xor     ax,ax            ;Zero ax
         xor     bh,bh            ;Zero bh
         mov     al,TEMPO ;Get the tempo weight
         test    al,80h           ;Is bit 7 set?
```

```
        jz      NO_NEG              ;No, it's a positive number
        neg     al                  ;It's negative, so make it positive
        xchg    al,cl               ;Want to divide duration by the weight
        div     cl                  ;Scale for specifeid duration
        xor     ah,ah               ;Get rid of the remainder
        jmp     short  DO_REST
NO_NEG:
        mul     cl                  ;Scale for specified duration
DO_REST:
        cmp     ax,100              ;Over one second?
        jb      UNDER_SEC           ;No, Under a second
        mov     ch,100              ;100 hundreds of a second
        div     ch                  ;Then get the seconds and 1/100's of sec
        mov     bh,al               ;Get the seconds
        mov     al,ah               ;Need 1/100's in al
UNDER_SEC:
        mov     bl,al               ;Get the 1/100's of seconds
        call    DELAY               ;and DELAY
        inc     si                  ;Next entry
        jmp     short  NOTE         ;Rest over, back to the melody
NO_REST:
        mov     al,[si]             ;Get the note duration
        inc     si                  ;Point to dot or to note
        cmp     byte ptr[si],'*'    ;Is it a dot?
        jne     NO_DOT              ;No, regular duration
        inc     si                  ;Point to possible double dot
        cmp     byte ptr[si],'*'    ;Is it another dot?
        jne     ONE_DOT             ;No, only one dot
        mov     bl,7                ;Multiplier for one and three fourths times
duration
        mul     bl                  ;Product into ax....the dividend
        mov     bl,4                ;Divisor for  one and three-fourths times
duration
        div     bl                  ;Double dotted note duration in al
        jmp     short  DOT_DONE     ;
ONE_DOT:
        dec     si                  ;Push back the table pointer
        mov     bl,3                ;Multiplier for one and a half times normal
duration
        mul     bl                  ;Product into ax which is the dividend
        mov     bl,2                ;Divisor into bl
        div     bl                  ;Dotted note duration in al
DOT_DONE:
        inc     si                  ;Point to note frequency
NO_DOT:
        mov     cl,al               ;Duration of note into cl
        xor     ax,ax               ;Zero ax
        xor     bh,bh               ;Zero bh
        mov     al,TEMPO            ;Get the tempo weight
        test    al,80h              ;Is bit 7 set?
        jz      POSITIVE            ;No, it's a positive number
        neg     al                  ;It's negative, so make it positive
        xchg    al,cl               ;Want to divide duration by the weight
        div     cl                  ;Scale for specifeid duration
        xor     ah,ah               ;Get rid of the remainder
        jmp     short  DO_DURATION
POSITIVE:
        mul     cl                  ;Scale for specified duration
DO_DURATION:
        cmp     ax,100              ;Over one second?
        jb      LESS_THAN_SEC       ;No, Under a second
        mov     ch,100              ;100 hundreds of a second
        div     ch                  ;Then get the seconds and 1/100's of sec
        mov     bh,al               ;Get the seconds
        mov     al,ah               ;Need 1/100's in al
LESS_THAN_SEC:
        mov     bl,al               ;Get the 1/100's of seconds
        mov     cl,OCTAVE           ;Get the octave weight
        mov     al,[si]             ;Get the note frequency
        mul     cl                  ;Note freq X octave = desired frequency
        mov     di,ax               ;Need frequency in di
        call    TONE                ;Produce the note
        inc     si                  ;Point to next entry
        jmp     NOTE                ;Go get it
PLAY_EXIT:
        ret

PLAY        endp


;Turn the speaker on and off

TONE        proc near

        mov     al,0b6h             ;Set up 8253 timer 2 for square wave
        out     43h,al              ; send it
        mov     dx,12h              ;8253 frequency of 11931802hz
        mov     ax,34deh            ; (1234de hex)into dx:ax
        div     di                  ;Divide by the note freq to get divisor
        out     42h,al              ;Send out the divisor...low byte
        mov     al,ah               ; then
        out     42h,al              ;      high byte
        in      al,61h              ;Get current speaker settings
        push    ax                  ;Save the settings
        or      al,3                ;Turn speaker on (bits 0 and 1)
        out     61h,al              ;Do it
        xor     ax,ax               ;No hours or minutes
        call    DELAY               ;DELAY for specified note duration
        pop     ax                  ;Recover speaker port settings
        out     61h,al              ;Turn speaker off
        ret

TONE        endp



;DELAY for a specified time interval

DELAY       proc near

        mov     ah,2ch              ;Get the current time
        int     21h
        xor     ax,ax               ;No hours or minutes
        add     ah,ch               ;Add current hours to specified hours
        add     al,cl               ;Add minutes
        add     bh,dh               ;Add seconds
        add     bl,dl               ;Add hundreds of seconds
        cmp     bl,100              ;Over one second?
        jb      SECONDS             ;No
        sub     bl,100              ;Yes, get all over one second
        inc     bh                  ;Add to seconds
SECONDS:
        cmp     bh,60               ;Over one minute?
        jb      MINUTES             ;No
        sub     bh,60               ;Get all over one minute
        inc     al                  ;Add to minutes
MINUTES:
        cmp     al,60               ;Over one hour?
```

Continued on Page 48

Getting the Most from Your Computer

Part 3

John Lewis
6 Sexton Cove Road
Key Largo, FL 33037

The focus of this series is to aid you in deriving the maximum utilization of your computer through exercising your imagination in the creation of significant programs. Pascal was named after Blaise Pascal, a French mathematician who said: "Imagination disposes of everything; it creates beauty, justice, and happiness, which are everything in this world". A rather profound observation and certainly one which bears scrutiny. My own experience of euphoria after the successful creation of a new algorithm or program segment tends to steer me in the direction of complete agreement with Pascal's statement.

In Part 2 of this series, we acquired the Pascal source code for a rather handy calculator which was added to the window/menu program from part one. In this article, our first task will be to convert the calculator code into a unit, making the inclusion of a calculator into some of your future programming efforts quite easy, possibly adding a touch of charisma.

Since we are building a program a step at a time and although each segment is a "stand alone" program, the final product is the sum of its parts, so let me address those readers who have just joined us. The best course of action would be to obtain copies of "REMark" that contain the previous two articles, failing that, send me a FORMATTED (3-1/2" or 5-1/4") disk and $5.00 for shipping and handling. My address is given at the beginning of this article. I will return your disk with the source code from the previous two articles.

The actual conversion of a working program segment into a unit is quite easy as you will soon see. Your first priority will be to make a copy of the source code for "CALC.PAS" and put the original in a safe place; you might otherwise become a victim of Murphy's law. You can use the DOS command: copy CALC.PAS CALCUNIT.PAS, assuming "CALC.PAS" is residing in the default directory. Now load Turbo Pascal and get ready to perform a little surgery to the file we have created under the name "CALCUNIT.PAS".

Remember, we discarded all the non-generic code from "DRAWBOX.PAS" when we made it into a unit. We will want to accomplish the same objectives here. The first consideration will be to change the first line of the program from "Program calc;" to "Unit calcunit;". This informs the compiler that it will be compiling the code into a unit ("CALCUNIT.TPU" (TPU - Turbo Pascal Unit)) rather than a program "exe" file.

There is no need to make any of the routines "Public" other than the one used to access the unit itself. In fact, the reverse is true, we should keep all the procedures and variables not needed by "calling" programs, private. Doing so will make a real contribution towards modularity. As a consequence, the code found under the "interface" heading of our new unit should be quite brief, mine consists of one procedure named: "Execute_calc". Where did "Execute_calc" come from?

Let's look at the code for "CALCUNIT.PAS" in Listing 1.

Go ahead and perform the required surgery on the code from Part 2 of this series to convert it into the unit shown. As you can see, only the removal of some selected pieces of code and the insertion of a Procedure name (Execute_Calc) along with a complementary "end", is required for the conversion. Be sure that you preserve the original source code for "CALC.PAS" when working on the unit version.

Our next step will be the creation of another "stand alone" program which will become item two on the main menu. I'm sure most of the readers who are following this series have a programming project that they would like to pursue, but first things first.

One subject continues to come to the attention of anyone working on computers and software. That is computer memory and the math required to address it. I know, you are probably thinking that I'm going to bore you with a lot of theory about binary and hexadecimal math. Not true. What we are going to do is create some software which will take much of the drudgery out of converting hexadecimal and binary numbers into decimal and vice versa. The conversion utility will not serve as a panacea, but will make any future dealings with number base changes much easier. It will also serve as a vehicle for learning about one of Pascal's math capabilities, integer arithmetic involving modulus division.

First, let's take a brief look at the number bases most frequently encountered in dealing with computer memory addressing, as well as the method used by the CPU in dealing with data and/or instructions. Hexadecimal (number base 16) is commonly used in the display of CPU registers, in disk file content representation (MS-DOS DEBUG uses this format) and numerous other places. In short, you will almost certainly encounter the hexadecimal numbering system in your pursuit of computer programming. Even more important (in my humble opinion) is the binary (base two) representation of numbers. When working with the logical operators (AND, NOT, OR, XOR, etc.) you will find that being able to see the bit locations of a given number will be a great help.

The hexadecimal numbering system uses the digits 0 - 9 and the letters A - F, for a representation of the numbers 0 -15 by a single character. The number 255 (decimal) has a hexadecimal counterpart FF. Dissecting each number in turn, we find that 255D (the trailing "D" denotes decimal notation) is equal to 5 (5 * 1) (units) + (5 * 10) (tens) + (2 * 100) (hundreds). FFH (the trailing "H" denotes hexadecimal notation) yields 15 (15 * 1) + 240 (15 * 16). The binary equivalent 11111111B (the trailing "B" denotes binary notation) yields 1 + 2 (1*2) + 4 (2*2) + 8 (2*2*2) + 16 (2*2*2*2) + 32 (2*2*2*2*2) + 64 (2*2*2*2*2*2) + 128 (2*2*2*2*2*2*2).

The first requirement in software design is the definition of the task to be performed, a rather simple chore in this instance. Our job is to design a program which will deal with each number system and display its equivalent in the other bases.

There are several methods available for converting decimal numbers into Hex, but we will use one that employs mathematical logic without any fancy tricks. We'll examine each step as we build our program, using the same number alluded to above in our discussion of number bases (255). If we divide 255 by 16, using the Pascal "Mod" (modulo) operator we derive a remainder of 15. Using fifteen (plus 1) as an index into the character array (string) "0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F", we find the character "F". We'll store that character for the time being in a Pascal string. Referring back to the division operation, we find that 255 "DIV" 16 also yields the number fifteen which becomes our second index into the character array alluded to above. Now we append the first character "F" to the second "F" for the hexadecimal number "FF", or more correctly, "FFH". If the second operation had yielded a number in excess of 15, we would have once again used the "MOD" operator on the dividend and continued as in the first case.

If the logic used here does not seem perfectly clear, I urge you to conjure up some other examples and subject them to the same algorithm. Notice too that I used numbers that would fit in one byte of computer memory for the sake of simplicity only. For instance: one added to 255 yields 256D or 100F, a number which must occupy two bytes (00000001 00000000 in binary). The algorithm works for larger numbers as well.

Now compare the above description to the actual program listing (DcHxUnit.pas) found in this article. Look closely at the procedure "Construct(val : Longint);" and compare the program logic to the description above. If I did a good job, you should

Listing 1

```
unit calcunit;
interface
procedure execute_calc;
implementation
uses boxunit, crt;

Var
   accum, val1  : Real;
   row, col, i : Byte;
   Option, Len : Integer;
   ch, chc, op : Char;
   s, st : string;

Function Get_result(s : String;p : char):Real;
   Var v : real;         { provide real storage for value of s }
   code : Integer;
   begin
   Val(s, v, code);      { get numeric value of string s, store in v }
   case p of
        '+':begin accum:=Val1+V;end;
        '-':begin accum:=Val1-V;end;    { compute result, store in accum }
        '/':begin accum:=Val1/V;end;
        '*':begin accum:=Val1*V;end;
        else accum:=V;
        end;
   Val1:=Accum;                { store accum in val1 }
   Get_result:=Val1;           { Function must be defined }
   end;

Procedure docalc;
   begin
      col:=2;row:=14;st:='';
      Accum:=0;ch:=chr(0);Val1:=0;    { initialize variables }
      gotoxy(col,row);
      while ch <> 'q' do              { provide means of escape }
         begin
         ch:=readkey;
         if ch <> chr(8) then st:=st+ch;  { enable backspace }
         if ch = chr(8) then
            begin
            if col > 2 then
               begin
               col:=col-1;gotoxy(col,row);  { update display after backspace }
               write(' ');
               end;
            len:=length(st);
            st:=copy(st,1,len-1);      { remove last char }
            end;
            write(ch);
         if ch <> #8 then
         col:=col+1;
         gotoxy(col,row);
         if ch = '=' then
            begin
               s:='';op:=chr(0);
               for i:=1 to length(st) do   { parse the input string }
                  begin
                     chc:=st[i];
                     case chc of
                     '0'..'9':begin s:=s+chc;end;
                     '.'     :begin s:=s+chc;end;
                     '*'..'+':begin st:=s;accum:=get_result(st,op);
                              s:='';op:=chc;end;
                     '-'     :begin st:=s;accum:=get_result(st,op);
                              s:='';op:=chc;end;
                     '/'     :begin st:=s;accum:=get_result(st,op);
                              s:='';op:=chc;end;
                     '='     :begin st:=s;accum:=get_result(st,op);
                              s:='';op:=chr(0);end;
                     end;
                  end;
                  gotoxy(col,row);write(' ',accum:4:4);{ format & display result
}
                  gotoxy(10,row+2);
                  write('Enter "q" to quit, any other key to continue ');
                  ch:=readkey;if ch <> 'q' then col:=2;gotoxy(col,row);clreol;

gotoxy(col,row+2);clreol;accum:=0;val1:=0;st:='';gotoxy(col,row);
            end;
         end;
      end;

Procedure Execute_calc;
   begin
      clrscr;
```

Listing 1 (Cont'd.)

```
      gotoxy(2,2);
      write('This calculator provides four functions using the following');
      gotoxy(2,4);
      write('keys: + (addition), - (subtraction), * (multiplication)');
      gotoxy(2,6);
      write(' / (division).');
      gotoxy(2,8);
      write('Problems must be entered in a number, operator, number format.');
      gotoxy(2,10);
      write('Pressing the equals key (=) will cause display of the answer.');
      gotoxy(2,12);write('Enter your problem: ');
      docalc;
   end;
end.
```

see a striking similarity.

Only two more steps remain in designing our program, that of displaying the decimal equivalent of a hexadecimal number and, finally, the binary representation of both.

Let's tackle the binary operation first. We will once again employ the "MOD" and "DIV" operators for our integer arithmetic and use decimal 30 for our example, applying much the same logic as in the decimal to hexadecimal problem.

We will repeatedly divide the original number (30, in this case) by 2, using first the "MOD" operator (yielding the remainder) and then the "DIV" operator (yielding the dividend) until the original number is reduced to zero. Applying this logic yields the binary number "11110", padding the result with three zero's for the sake of clarity (since we are dealing with 8 bit bytes) gives us "00011110". Success!

Once again, look at the listing for DCHXUNIT.PAS and compare the logic in "Procedure Binary(val : Longint);" with the explanation just given. Hopefully, you will see a close similarity. I apologize for the extra complexity caused by the need for display of numbers too large for a single byte representation; however, this portion of the program is quite effective in providing an attractive (and useful) display of its binary output and that, after all, is adequate justification for some extra code.

Before we proceed to the hexadecimal-decimal conversion, let's look at some routines found in "Procedure Binary(Val :Longint)", which deserve a little extra attention: "while" and "repeat". The "while" routine is a "loop control" method that continues execution as long as the condition specified is TRUE. All of the statements within the "begin" and "end" markers are performed in sequence until the conditional statement becomes FALSE. If it is False to begin with, no execution occurs.

A similar statement, but different in one very important aspect, is the "repeat until" routine. This statement will cause repeated execution of the statements within the "begin" and "end" delimiters until the specified condition becomes FALSE. One iteration of the loop always occurs, regardless of its conditional state.

Now we can attack the hexadecimal to decimal conversion. It poses a little different problem in that any given hexadecimal entry may contain some characters which have no numeric (decimal) value (A - F). The most logical way to deal with this anomaly is to use a "lookup table". Ours already exists in the form of "Hexstring". We can compare a given (starting with the least significant) character in the input string with each character in "Hexstring", incrementing the value of our reference until a match is found. The reference will thus attain a decimal value corresponding to its hexadecimal counterpart. By multiplying with a factor equal to 16 times its less significant neighbor, we can assemble the decimal equivalent of our hexadecimal entry.

If you examine the "Procedure Hex_ToDec(HexStr : String20)"(within "DcHxUnit.pas"), you will, hopefully, see a parallel to the above. Before we continue, let's look at one area of this routine which might cause a bit of confusion. We use a double "while" statement, each containing a parameter for comparison. Bear in mind that whenever you encounter this condition, the innermost "while" loop executes first, then the next outermost. This technique can be used with several "levels" of while loops, each one performing its task before "falling through" to the next. In this case, the procedure designates "Found" as equal to the character within "Hexstring" (indicated by the value of its string index plus one). Next, "Found" is compared to the character within the input string that corresponds to the index: "Len" (length of string). "I" (the Hexstring index) is incremented with each iteration until a match is found ("ch <> Found" becomes false).

The procedure now "falls through" to the next "while" statement (while Len > 0). The variable "Dec" is made equal to the index used above, multiplied by a factor which starts with 1 and is incremented by 16 with each iteration. The result is a decimal number, equal to its hexadecimal equivalent. See Listing 2.

I hope that you will study the program listing for DCHXUNIT.PAS until you understand it fully. It contains some logic that you might find useful in other programming projects. There are no surprises in the code, just some straightforward logic used to solve a problem.

Notice that you have been working on the code listing for a Pascal "unit" rather than a program. I decided that we could graduate to constructing a unit directly without going through the intermediate step of coding a program and then modifying the file to construct a unit. The conversion process is largely redundant after all.

I promised you an added item for the Window/Menu program from Part 1 of this series. All we have to do now to accomplish the inclusion of this unit in our project program is a bit of painless surgery. Let's look at Listing 3 for "PROJCT_1.PAS".

Notice that nearly all the extraneous code has been culled from our project program and it has undergone a name change. Only the bare essentials remain.

Listing 2

```
Unit DcHxunit;
Interface
Procedure Execute_DcHx;
Implementation
uses Crt, Dos;
Type
   String20 = String[20];
   String1  = string[1];
      const
      HexString : String20 = '0123456789ABCDEF';

Var
   OutString, InString, Dum : String20;
   St : String1;
   Value, Dec    : Longint;
   Dumy : Integer;
   Ch : Char;

Procedure Binary(Val : Longint);
   Var len, Byte_Count, Format : Integer;
   Begin
      Len:=0;Format:=8;
      Outstring:='';If val < 256 then Byte_Count:=1     { no ";" when using else }
      Else if (val > 255) and (val < 65536) then
         Begin
            Byte_Count:=2;
            Format:=17;
         end;
      While Val > 0 do
```

Listing 2 (Cont'd.)

```
        Begin
           str(val mod 2,st);              { get val modulo 2, convert to string }
           outstring:=st+outstring;        { append outstring to st }
           Val:=Val Div 2;                 { integer division by 2 }
           len:=len+1;                     { add to length of outstring }
           if len = 8 then                 { check for end of byte }
           outstring:=' '+outstring;       { tack on a space }
        end;
           If Length(Outstring) < Format then  { check for length }
        repeat
           begin
              insert('0',outstring,1);        { Pad the output string with zeros }
              len:=length(outstring);
           end;
        until len = Format;                { are we done ? }
   end;

Procedure Hex_ToDec(HexStr : String20);
   Var Len, I : Integer;Count : LongInt;
   Found : Char;
   begin
      Len:=Length(HexStr);I:=0;Dec:=0;Count:=1;
      while Len > 0 do begin
         while ch <> Found do
         begin
            Found:=HexString[I+1];         { I = index into Hexstring }
            ch:=HexStr[Len];I:=I+1;
         end;
```

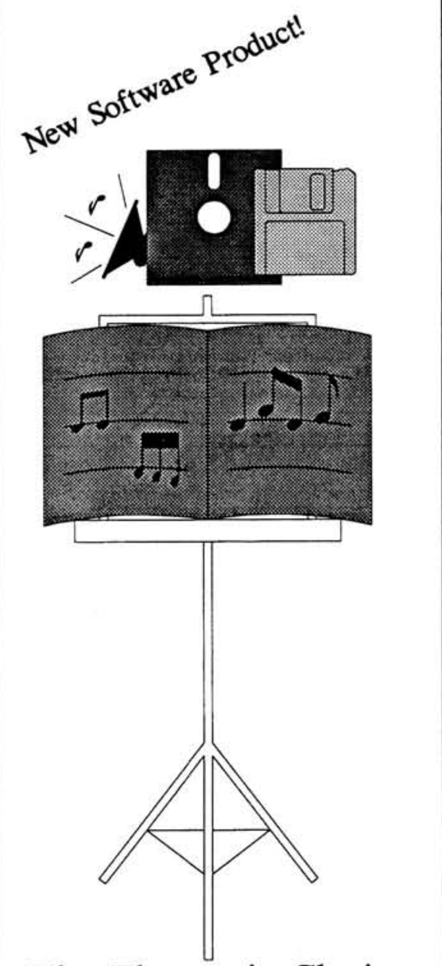
The three "units" house nearly all of the more sophisticated routines.

If you are new to programming, I would urge you to use this new program segment to view a graphic representation of Hexadecimal and Binary numbers and their correlation to each other. For instance, Hex "F0" and its binary equivalent. A few minutes spent using this program can give you quite an insight into how the two number bases complement each other.

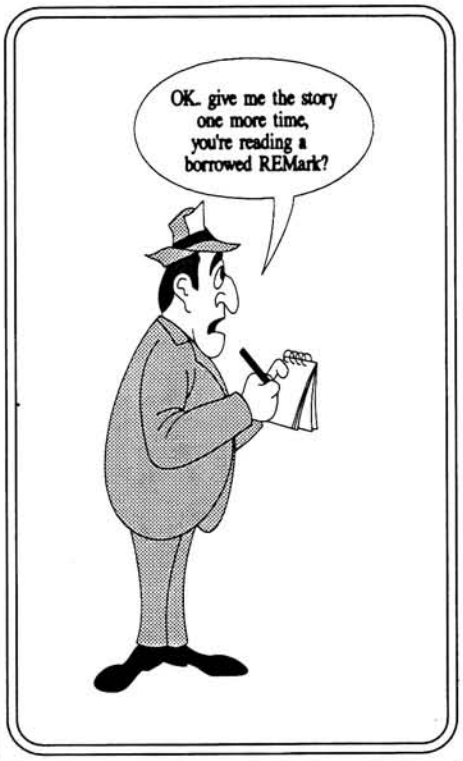
If you have any questions regarding this, or any of the articles which comprise this series, or even Turbo Pascal programming in general, please feel free to write me at the address given at the beginning of this article. Be sure to enclose a S.A.S.E. (Self addressed, stamped envelope) if you wish a reply. Feel free to make comments about the series. If you stipulate that your letter may be included in a future article, that would be helpful.

Listing 3

```
Program Projct_1;
uses boxunit, crt, calcunit, DcHxUnit;

Var
option : integer;

Procedure Esthetics;
   begin
      textcolor(black);textbackground(lightgray);
   end;

Procedure SetUp;
   Begin
      window(Left+1,Top+1,Right-1,Bottom-1);
      TextBackground(Blue);TextColor(Yellow);clrscr;
   end;

begin
   while option <> 6 do
   begin
      clrscr;Esthetics;
      drawbox(left, top, right, bottom);        { uses Boxunit }
      setup;                {top=4, left=5, right=75, bottom=22 }
      gotoxy(11,2);write('Please enter...');
      gotoxy(20,4);write('1. Calculator');
      gotoxy(20,6);write('2. Decimal, Hexadecimal, Binary conversion.');
      gotoxy(20,8);write('3. Function not yet implemented.');
      gotoxy(20,10);write('4. Function not yet implemented.');
      gotoxy(20,12);write('5. Function not yet implemented.');
      gotoxy(20,14);write('6. Exit to operating system');
      gotoxy(18,16);write('the number of your choice ');
      readln(option);clrscr;
      case option of
         1:begin Execute_calc;end;    { uses calcunit }
         2:Begin Execute_DcHx;end;    { uses DcHxUnit }
         end; { end case option of }
   end;  { end of while }
end.
```

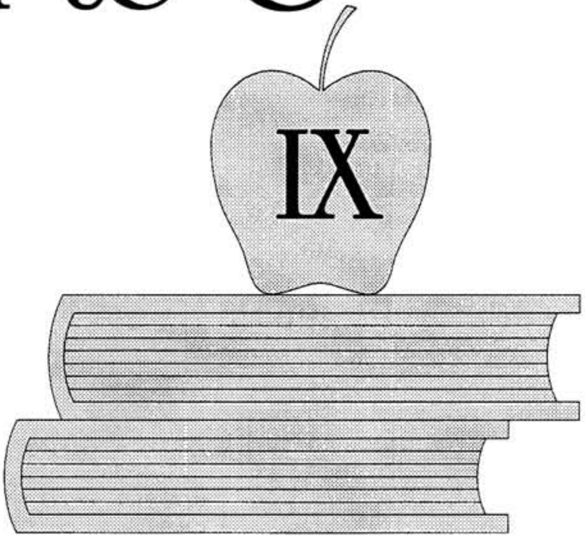


The Electronic Clavier
P/N 885-6016


OK. give me the story one more time, you're reading a borrowed REMark?

Introduction to C++

Ninth Installment



Lynwood H. Wilson
2160 James Canyon
Boulder, CO 80302

Resources

I have installed the new Borland C++ version 2.0 which recently appeared. At first I intended to stick with the Turbo C++ since I do not write software to run under windows yet, but my spies told me that there was a lot more to the Borland compiler than just windows. And there is.

Borland intends the Turbo C++ compiler to be their low end tool. The Borland compiler will be the top of the line, the developer's tool. In just the two days since I installed it, I am convinced. It compiles faster, runs in protected mode so it can handle larger programs, and it produces much smaller executable programs. They say the editor is much improved also, but I still use Brief.

Since last month I have begun exploring the Blaise Turbo C Tools. I was looking for a way to do menus and pop-up dialog boxes with mouse support in character mode quickly, without too much work. There is, of course, no free lunch but I was hoping to find a moderately priced meal. I found it.

The Blaise Tools include over 200 functions which do a lot of useful things. Menus and windows with mouse support, help screens, string functions, a tiny editor to use in your programs, and plenty more. And the best part is that the functions run from high- to low-level. It's as though they gave us not only the finished high-level functions they thought we would find useful, but also all the tools they used to build them, in case we wanted to do it differently. This answers the main objection I have always had to packages like the Zinc Interface Library I mentioned last month; which is that they are great if you want to do exactly what the designer of the package wanted to do, but they lack flexibility. The Blaise tools are flexible. And they give you the source, at no extra charge. And you can read their book. Recommended.

In three days I was able to cobble together a demo of the whole menu system for the program I am working on, including a few sample data entry screens. Much of this was made from the Blaise demo code. The client was properly impressed.

As I've gotten deeper into the program, I've run into the usual conflicts between the way Blaise did it and the way my customer wants it. In each case, I have been able to drop down a level to more primitive functions from Blaise and Borland, and get it just like I want it. And don't overlook the value of being able to show the client something close to what he wants very quickly, even if you have to change it later.

The only minor problem is that the Blaise Tools are written in C, not C++. It is easy enough to use them (or any other C functions) in a C++ program. You need only to tell the C++ compiler that the functions are straight C. This is most easily done by making the entire header files which contain the prototypes for the C functions extern "C", like this:

```
extern "C" {
   #include <bscreens.h>          // Blaise
   #include <bmouse.h>            // headers
}
```

If you only have a few C functions, you can add the extern "C" to their prototypes.

```
extern "C" void c_func(char ch);
```

The reason this is necessary is that C++ appends to the name of each function a series of characters which describe the parameters the function takes. It's called name mangling. This is how the compiler can do data type checking on parameters to functions from different disk files. By declaring functions to be "C", we tell the compiler that they will not have these appendages, and therefore, data type checking must be suspended for these functions.

So it is quite easy to use the Blaise Tools with C++. However, the tools would be the better for having been written in C++ and I expect Blaise is working on it. This seems to me to be the area where OOP does us the most good, in making code easily reusable in new programs. I found myself making several of the Blaise Tools into objects just because that is the way I am beginning to think. And I found that the effort paid off each time I used them.

I will talk more about that and write up some of the code in a future article. Right now, we'd better finish up with pointers.

More Pointers

Several installments ago, we learned how to initialize a character array with a string like this:

```
char agreeting[] = "Hello, world";
```

A similar thing can be done with a char pointer, like this:

```
char *pgreeting = "Hello, world";
```

From the similarity of these statements and what we have seen of the similarities between array and pointer notation, these two look a lot alike. They are both used often, but here are differences worth mentioning.

First, agreeting and pgreeting are both char pointers, since they both hold the address of a character; but pgreeting is a pointer variable and agreeting is a pointer constant. Remember that agreeting holds the address of the beginning of the array which holds the string. (Watch the distinction here between the memory and what is stored in it. Agreeting is a character array which holds the string "Hello, world".) In order that it always point to the head of the array, it cannot be changed. Thus, it is a constant. Pgreeting also points to a char which is the beginning of a string, but it can hold other values, can be made to point to other characters as well.

There is another important difference. When we define the array agreeting, we allocate memory and store the initial chars in it. Later in the program we can store other chars in that memory, since it is an array of char variables. In the other case, the string of chars whose address is assigned to pgreeting form a string constant. The result of an attempt to modify a string constant is undefined. It is not an array of char memory which you may use as you like.

The values of agreeting and pgreeting can both be passed to functions as char pointers, but the functions must observe the differences between the things pointed to. By passing the value of agreeting to another function, or by assigning it to a char pointer variable in the same function, we can overcome the limitation that we cannot change the value of agreeting (the pointer). Thus, see Figure 1.

```
#include <iostream.h> void pr_str(char *pstr); void main(void) {
   char agreeting[] = "Hello, world";
   char *pgreeting = "Hi yourself";
   pr_str(agreeting);
   pr_str(pgreeting);
}

void pr_str(char *pstr) {
   while(*pstr)
      cout << *pstr++;
   cout << '\n';
}
```

Figure 1

In the function, both the strings look the same and they both get printed to the screen. However, they are not the same and we must remember it, even if the function cannot tell the difference. For example, it is perfectly reasonable for us to copy the string pointed to by pgreeting to the array agreeting, but we must not copy anything to the memory pointed to by pgreeting since that memory holds a string constant.

Note that we can pass the name of an array to a function and declare it in the function as a pointer. This is legitimate since the array name is a pointer to the beginning of the array. Pointer and array notation can be mixed as you like, the main restraining factor being the necessity to read the code again someday.

Note also that the "while" loop that prints the string a character at a time could be replaced by

```
cout << pstr;
```

because "cout" knows (from the definition) that pstr is a char pointer and will print the whole string, up to the NULL.

And note yet again that we could have defined the first parameter to the function as an array without changing anything else. In other words, the first line of the function could have been:

```
void pr_str(char pstr[])
```

This may seem wrong because we are incrementing the value of pstr in the function and I told you that you could not change the value of a pointer constant, such as the name of an array. However, no matter which way we define the parameter we pass into the function, it is still a pointer to an array, not the actual name given to the array at the time the memory was allocated, and so you can change its value as you like.

Arrays of Pointers

When you need a set of strings, for use in a menu or something, it is convenient to be able to bunch them together into a single entity. We already know how to do that with a multi-dimensional array, but the more common method uses an array of pointers. If you watch closely, you will notice that although this works a lot like a two-dimensional array, there are significant differences.

Here is a declaration of an array of 5 pointers to char.

```
char *pch[5];
```

Note that these do not yet point to anything (that we know of). We can make them point to a string just as we did with a single pointer above.

```
pch[2] = "This string";
```

Remember, pch[2] is just a plain char pointer just like pgreeting. If we'd like to initialize the whole batch of pointers at the same time, we can do it like this:

```
#include <iostream.h>
void pr_items(char **pstr, int n);
void main(void) {
   char *items[] = { "One item",
                     "Another item",
                     "Yet another",
                     "The last"     };

   pr_items(items, 4); }

void pr_items(char **pstr, int n) {
   for(int i = 0; i < n; i ++) {
      cout << pstr[i];
      cout << '\n';
   }
}
```

Note that "items" is an array of char pointers which are initialized with the addresses of the four strings. It is not an array of strings (a two-dimensional array of char) The strings are not stored in items, their addresses are.

Note also the declaration of the first parameter in pr_items. pstr is a pointer to a pointer to a char. This is sometimes called double indirection. In the body of the function, I switched back to array notation, and sent pstr[i] to cout. The string is printed because pstr[i] is a pointer to char, as described above.

This array of pointers to strings saves a bit of memory compared to using a two-dimensional array, since the second dimension of the array would have to be big enough to hold the longest string. Since the pointers in our array point to string literals or constants, each takes only the space it requires.

This operation would be more convenient if we did not have to pass the size of the array to the function. If you'll recall, we solved that problem in the case of strings by ending with a unique value which could not be normal data, and which tells us that we are at the end. We can do the same here.

```
#include <iostream.h> void pr_items(char
              **pstr); void main(void) {
   char *items[] = { "One item",
                     "Another item",
                     "Yet another",
                     "The last",
                     ""
                     };

   pr_items(items);
}

void pr_items(char **pstr) {
   for(int i = 0; *pstr[i]; i ++) {
      cout << pstr[i];
      cout << '\n';
   }
}
```

In this example, the last string is an empty or NULL string with nothing in it except the NULL. In the function, the for loop runs until the char pointed to by pstr[i] is NULL. Thus, we do not have to pass in the length of the array.

You can accomplish the same thing by making the value of the last pointer NULL, rather than by making it point to a NULL.

```
#include <iostream.h> void pr_items(char
              **pstr); void main(void) {
   char *items[5] = { "One item",
                      "Another item",
                      "Yet another",
                      "The last"
                      };
   items[4] = NULL;

   pr_items(items);
}

void pr_items(char **pstr) {
   for(int i = 0; pstr[i]; i ++) {
      cout << pstr[i];
      cout << '\n';
   }
}
```

In this example, we had to explicitly assign the value NULL to the last pointer. This is not quite as convenient in this case, but we will see cases later where it is more so. Note that in this function we end the loop when pstr[i] itself is false, rather than that which it points to.

Note that when we are assigning NULL to a pointer we use the symbol NULL rather than the number 0. NULL is 0, but not all zeros are equal. Null is defined for us in several different header files, and its value is 0 if we are in any of the smaller memory models in which pointers are two bytes long and 0L in the larger memory models in which pointers are four bytes.

In old C, programmers often assumed that pointers were integers. It worked when they had less than 64K bytes of memory, because the pointers were the same as integers. Such assumptions must be purged from the programs before they can be ported to systems using larger memory models. We should not trap ourselves by doing the same sort of things.

You might use such an array of pointers to strings or other data types if you were going to sort the data. It is much quicker to move the pointers around in their array than it is to rewrite a long string or some larger data type.

Void Pointers

So far we have seen that pointers are always defined as pointing to a particular data type. Occasionally, however, a situation will arise in which you will need a pointer, but you don't want to commit yourself at compile time as to what it will point to. The solution is a void pointer. Void pointers cannot be dereferenced or be used in pointer arithmetic because these things won't work unless the compiler knows what kind of data they point to. However, a pointer of any other type can be cast to type void and back again without loss of information.

If you declare a function like this:

```
void ambiguous(void *ptr);
```

you can pass to it a pointer of any kind. If you had declared pointers like these:

```
char *pch;
int *pi;
float *pfl;
```

you could then call the function like this:

```
ambiguous(pi);
ambiguous(pch);
ambiguous(pfl);
```

in the main code. The function would have to know, somehow, what kind of pointer each of them was, perhaps by another parameter which acted as a flag. This is similar to function overloading, but not quite the same.

When you use overloaded functions, the compiler figures out which of several functions with the same name to use by the data types and number of arguments. Note that this is done at compile time. If you don't know in advance because it depends on the input to the program, and you have to decide at run time, then overloaded functions will not help. Void pointers can be the solution.

Pointers to Functions

As in the case of arrays, the name of a function holds its address. Given this address, we may pass a function to a function as a parameter, and assign the address to a pointer which points to the function. We can call the function just as if we knew its name. Here is a simple example.

```
#include <iostream.h>
void print_int(int x);
void print_2int(int x);
void a_func(int y, void (*pf)(int x));
void main(void)
{
   int a = 17;

   a_func(a, print_int);
   a_func(a, print_2int);
}

void print_int(int x)
{
   cout << x << '\n';
}

void print_2int(int x)
{
   cout << 2 * x << '\n';
}

void a_func(int y, void (*pf)(int x))
{
   pf(y);
}
```

The functions print_int() and print_2int() are simple functions that take an int argument and return nothing. A_func() takes as its arguments an int and a pointer to a function that takes an int argument and returns nothing. Here is the definition of that pointer.

```
void (*pf)(int x)
```

It might seem more reasonable to do it like this:

```
void *pf(int x)
```

but if you think of it, this is the definition of a function which takes an int argument and returns a void pointer. The parentheses around the name and the asterisk make it into the definition of a pointer instead.

To call a function using a pointer to it, you can treat the pointer to the function as though it were the function's actual name. It also works to dereference the pointer explicitly and call the function like this:

```
(*pf)(x);
```

but that is exactly like saying:

```
pf(x);
```

As in the case of arrays, since the name of the thing holds the address of the thing, we may use the pointer to the thing as though it were the name of the thing.

Command Line Arguments

It is often useful to pass arguments to a program when you execute it. When I started this editor, for instance, I added the name of the file I wanted to edit. When you add arguments to the program name on the command line, the operating system passes them to the program as two arguments to the program. The first is an int which holds the total number of arguments, and the second is a pointer to an array of char pointers to the argument strings themselves. Here is a simple example which prints out all the command line arguments.

```
#include <iostream.h> void main(int argc,
                                char *argv[]) {
   for(int i = 0; i < argc; i++)
      cout << argv[i] << '\n';
}
```

The first argument is always the name of the program itself, with the full pathname. After all, it is the first thing on the command line. You might argue that the program should know its own name, but anyone can rename it and, in any case, it can sometimes be useful for a program to know where it lives.

The names of the two arguments, argc and argv, are not mandated by the language, but they are the same ones used by everyone since Kernighan and Ritchie themselves and it's a pretty good practice to stick to them.

The standard requires that the list of pointers to arguments, argv, be terminated by a NULL pointer. Thus, we could have written the program like this:

```
#include <iostream.h> void main(int argc,
                                char **argv) {
   while(*argv)
      cout << *argv++ << '\n';
}
```

This version prints arguments until the next pointer in the array of pointers to strings has the value NULL. The only problem is that the program doesn't need the other argument, argc, and we get a warning from the compiler that it is not being used. If we leave the argument out of the program it doesn't work. However, if we leave out the variable name and keep the data type everything comes out fine and we don't get the warning message. Like this:

```
void main(int, char **argv)
```

This is a useful technique in all the cases where you have unused arguments for any reason. It is dangerous to get into the habit of disregarding warnings, better to get into the habit of writing code that does not produce warnings. It's not always possible, but we can try.

References

As we have seen, the arguments passed to functions in C and C++ are passed by value, rather than by reference. The function gets a copy of the value of the parameter to use as it likes, and the original variable back in the calling code is unchanged.

We can choose to pass the function a pointer to a variable in the calling code to allow the function to change that variable, or to save memory, but the responsibility is on us to pass the address correctly and to make sure the function and the call match. This isn't difficult, as we've seen above, but it could be more convenient. Here is how to make it more convenient.

A reference in C++ (no such thing in C) is like a pointer in that it contains the address of something else, and like an ordinary variable in that you need not dereference it. Their main use is in passing variables to functions. Here is the switch_em program from last month, rewritten to use references rather than pointers.

```
#include <iostream.h> void switch_em(int
        &a, int &b);  void main(void)  {
   int x = 17, y = 23;

   cout << "\nX = " << x;
   cout << "\nY = " << y;
   switch_em(x, y);
   cout << "\nX = " << x;
   cout << "\nY = " << y;
}

void switch_em(int &a, int &b)  {
   int temp;
   temp = a;
   a = b;
   b = temp;
}
```

In the definition and the declaration (prototype) of the function switch_em(), the arguments a and b are references. They are declared int &a, which is the declaration of a reference.

Since references need not be dereferenced with the * like pointers, they are used directly just like simple variables in the function. You can think of this construction as a way of passing parameters by reference; passing the actual parameter instead of its value, merely by putting ampersands (&) in front of the variable names in the function definition and declaration. There is no change in the calling code, it looks just like it would if you were passing ordinary parameters in the ordinary way. The difference is all in the function.

Herein lies one of the disadvantages of using references. You cannot tell from reading the calling code that the function can change the values of the variables in the calling code. I know of no solution to this potential error, and therefore, I recommend careful and limited use of references.

As with pointers, references are often used when the intention is to save memory by not creating another copy of a large data structure. In this case, it is good practice to declare your intention of not changing the value of the parameter by making it a constant.

```
void foo(const int &bar);
```

Here "bar" is a reference to an integer argument being passed in, but it is a constant in the function (regardless of its status in the calling code) so that the value of bar may not be changed.

Note that even though a reference is a little like a pointer, it cannot be changed after it is initialized. It cannot be made to point to something different.

There can be no references to references, no arrays of references, and no pointers to references. However, you can take the address of a reference, in which case you get the address of the thing referenced.

Independent References

References can also be used independently of function calls. Here is such a reference definition.

```
int x;
int &rx = x;
```

Such an independent reference, like all references, must be initialized when it is created. The reference rx is an alias for x and can be used in place of x in any context. However, it offers no advantage over using x directly, and will potentially make the code harder to read. I cannot think of any reason for doing this.

Sources

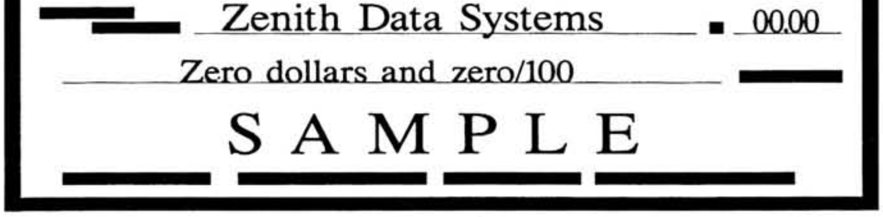
C Tools Plus	$149.00 (includes source)

Blaise Computing Inc.
2560 Ninth Street Suite 316
Berkeley, CA
(800) 333-8087

Continued from Page 40

```
          jb       HOURS          ;No
          sub      al,60          ;Yes, get all over one hour
          inc      ah             ;Add to hours
HOURS:
          cmp      ah,24          ;Over 24 hours?
          jne      CHECK          ;No, check elaspsed time
          sub      ah,ah          ;Yes, get rid of excess
CHECK:
          push     ax             ;Save hours and minutes
          mov      ah,2ch         ;Get current time
          int      21h
          pop      ax             ;Get hours and minutes
          cmp      cx,ax          ;Do they match?
          ja       QUIT           ;No,time's up!
          jb       CHECK          ;No, check again
          cmp      dx,bx          ;Yes, how about seconds and hundreds?
          jb       CHECK          ;Not yet
QUIT:                             ;Yes, time's up
          ret

DELAY     endp

          end
```

All Checks must be made out to
Zenith Data Systems


January 1, 1991
Zenith Data Systems 00.00
Zero dollars and zero/100
SAMPLE

Is lack of computer knowledge holding you up?

"Powering Up" could be your best protection!

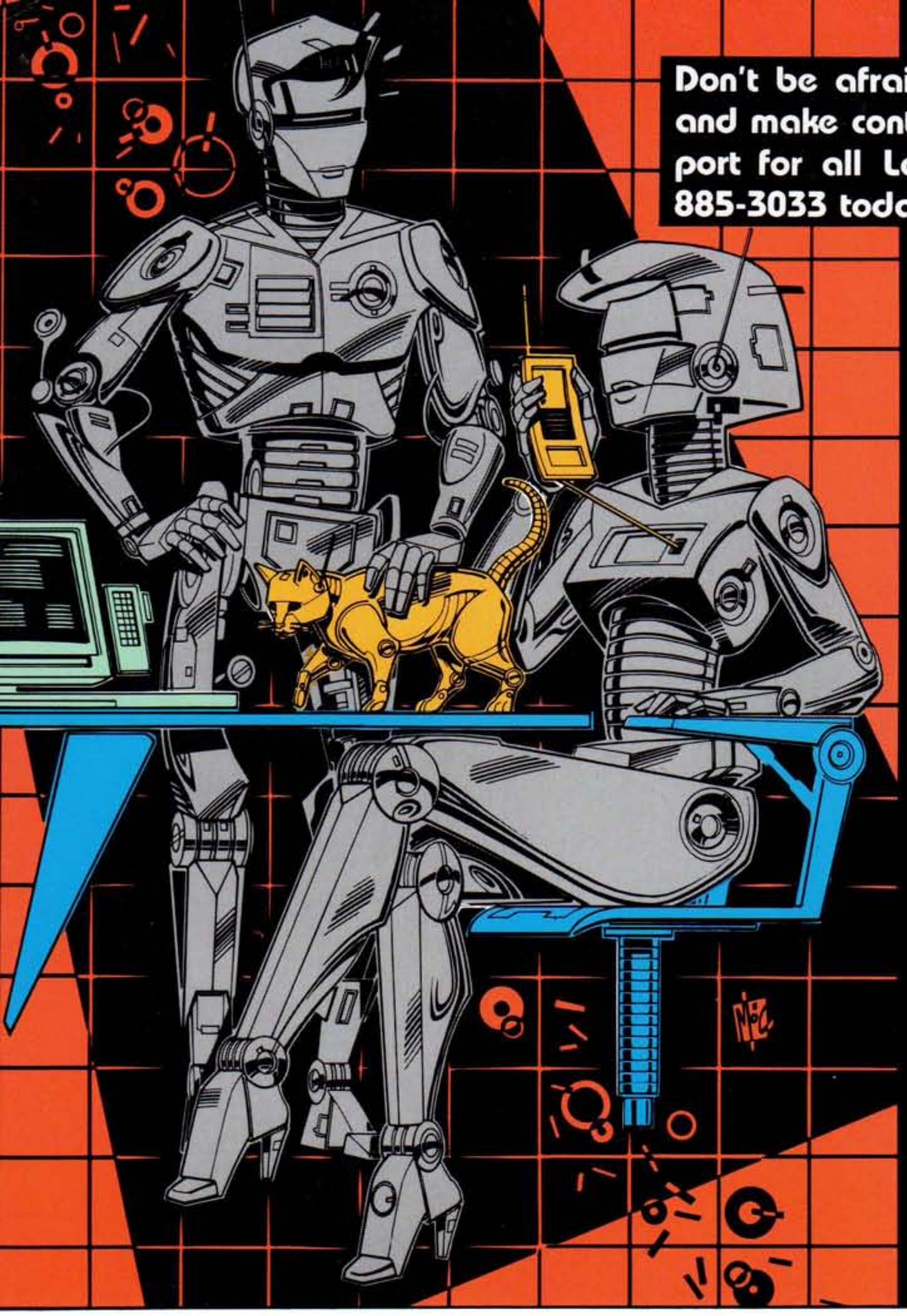
"Powering Up"
by Bill Adney
Order today!
ZUG (616) 982-3463

HADES II

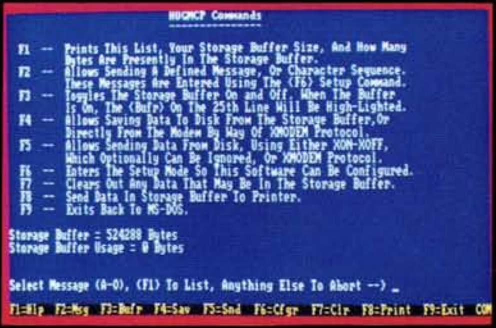
It's HOTTER than ever! Jam-packed with new features, HADES II still remains the easiest-to-use disk editor ever! Just look at some of the features:

- Sector Display/Editing
- Sector HEX/ASCII String Search
- File Display/Editing
- Physical and Logical Cluster Display
- File HEX/ASCII String Search
- Drive Parameter Display
- 512 MegaByte Drive Size Limit
- File Attribute Display/Edit
- Automatic Erased File Recovery
- Manual Rebuild File Recovery
- Works with Headerless MS-DOS Disks
- PC-Compatible or H/Z-100

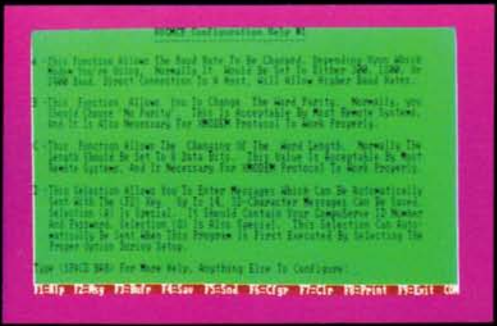
HADES II is still only $40, and original HADES owners can upgrade their distribution disk for only $15. Call HUG today at: (616) 982-3463.



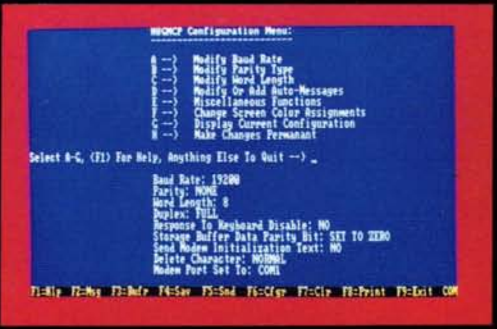
Don't be afraid to communicate! Get HUGMCP and make contact the easy way. Now with support for all Laptops, order HUG Part number 885-3033 today.



```
                      HUGMCP Commands
                      ---------------

F1 --  Prints This List, Your Storage Buffer Size, And How Many
       Bytes Are Presently In The Storage Buffer.
F2 --  Allows Sending A Defined Message, Or Character Sequence.
       These Messages Are Entered Using The (F6) Setup Command.
F3 --  Toggles The Storage Buffer On and Off. When The Buffer
       Is On, The (Bufr) On The 25th Line Will Be High-Lighted.
F4 --  Allows Saving Data To Disk From The Storage Buffer,Or
       Directly From The Modem By Way Of XMODEM Protocol.
F5 --  Allows Sending Data From Disk, Using Either XON-XOFF,
       Which Optionally Can Be Ignored, Or XMODEM Protocol.
F6 --  Enters The Setup Mode So This Software Can Be Configured.
F7 --  Clears Out Any Data That May Be In The Storage Buffer.
F8 --  Send Data In Storage Buffer To Printer.
F9 --  Exits Back To MS-DOS.

Storage Buffer = 524288 Bytes
Storage Buffer Usage = 0 Bytes

Select Message (A-O), (F1) To List, Anything Else To Abort --> _
F1=Hlp  F2=Msg  F3=Bufr  F4=Sav  F5=Snd  F6=Cfgr  F7=Clr  F8=Print  F9=Exit  COM
```



```
                  HUGMCP Configuration Help #1
                  ----------------------------

A -This Function Allows The Baud Rate To Be Changed, Depending Upon Which
   Modem You're Using. Normally It Would Be Set To Either 300, 1200, Or
   2400 Baud. Direct Connection To A Host, Will Allow Higher Baud Rates.

B -This Function Allows You To Change The Word Parity. Normally, you
   Should Choose 'No Parity'. This Is Acceptable By Most Remote Systems,
   And It Is Also Necessary For XMODEM Protocol To Work Properly.

C -This Function Allows The Changing Of The Word Length. Normally The
   Length Should Be Set To 8 Data Bits. This Value Is Acceptable By Most
   Remote Systems, And Is Necessary For XMODEM Protocol To Work Properly.

D -This Selection Allows You To Enter Messages Which Can Be Automatically
   Sent With The (F2) Key. Up To 14, 32-Character Messages Can Be Saved.
   Selection (A) Is Special. It Should Contain Your CompuServe ID Number
   And Password. Selection (O) Is Also Special. This Selection Can Auto-
   matically Be Sent When This Program Is First Executed By Selecting The
   Proper Option During Setup.

Type (SPACE BAR) For More Help, Anything Else To Configure:
F1=Hlp  F2=Msg  F3=Bufr  F4=Sav  F5=Snd  F6=Cfgr  F7=Clr  F8=Print  F9=Exit  COM
```



```
              HUGMCP Configuration Menu:
              --------------------------

              A -->   Modify Baud Rate
              B -->   Modify Parity Type
              C -->   Modify Word Length
              D -->   Modify Or Add Auto-Messages
              E -->   Miscellaneous Functions
              F -->   Change Screen Color Assignments
              G -->   Display Current Configuration
              H -->   Make Changes Permanant

Select A-G, (F1) For Help, Anything Else To Quit --> _

              Baud Rate: 19200
              Parity: NONE
              Word Length: 8
              Duplex: FULL
              Response To Keyboard Disable: NO
              Storage Buffer Data Parity Bit: SET TO ZERO
              Send Modem Initialization Text: NO
              Delete Character: NORMAL
              Modem Port Set To: COM1
F1=Hlp  F2=Msg  F3=Bufr  F4=Sav  F5=Snd  F6=Cfgr  F7=Clr  F8=Print  F9=Exit  COM
```

ZENITH
data systems
Groupe Bull

BULK RATE
U.S. Postage
PAID
Zenith Users' Group

POSTMASTER: If undeliverable, please do not return.

$2.50
P/N 885-2139