MicroDot

The logical replacement for BASIC.SYSTEM

For Apple //+(w/64k), e, c, gs

-ProDOS based -

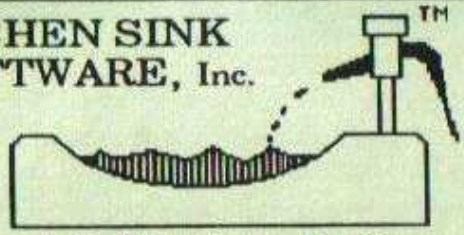
Uses standard ProDOS files

Less Than 4K In Size

Replaces BASIC.SYSTEM (10.6K long)

You Gain 7K+ of Extra Program space!

KITCHEN SINK SOFTWARE, Inc.™



903 Knebworth Ct.
Westerville, OH 43081
(614) 891-2111

Special features:

* Automatic BASIC overlays.
* ProDOS disk formatter.
* Pack HiRes graphic screens direct to disk.
* Access Auxtype bytes.
* Works with *Program Writer*
* Much more!

MicroDot

The logical replacement for BASIC.SYSTEM

By: Jerry Kindall

If you do not already have one...

SEND FOR OUR

FREE CATALOG!

KITCHEN SINK SOFTWARE, Inc.™

903 Knebworth Ct.
Westerville, OH 43081
(614) 891-2111

Special thanks to Mark deJong & Alan Bird of Beagle Bros. for assisting with the Program Writer patch and also to David Ely of Palace Productions for extensive testing.

TABLE of CONTENTS

Getting Started

Updates

Naturally, we will always be looking for ways to improve all of our software products, including *MicroDot*. To get an update, send us the original disk and a small fee. Call us for the current cost when you want to update .

Licensing *MicroDot*

Having purchased *MicroDot*, you can use it all you want for your personal use. If you wish to distribute or sell disks that use MicroDot, it is easy to get a license to do so. Just write to us for details. There is a small one time fee for the license.

Routines others wrote and we Licensed (after modifying)

The **FORMAT** programs are based on Jerry Hewett's public domain HyperFORMAT routine, which is distributed on Misk Disk #1 from Living Legends Software.

The **Hires Packer/unpacker** is based on Polarware's packer. Though we modified it a lot we wish to give credit where credit is due:

Graphics routines from The Complete Graphics System by Polarware were written by Mark Pelczarski, Steven Meuse, David Lubar, and David Shipiro, and are copyrighted 1984 by Polarware. All rights reserved. The Complete Graphics System and Polarware/Penguin Software are trademarks of Polarware.

Apple Computer, Inc. makes no warranties, either express or implied, regarding the enclosed software package, its merchantability or its fitness for any particular purpose. The exclusion of implied warranties is not permitted by some states. The above exclusion may not apply to you. This warranty provides you with specific legal rights. There may be other rights that you may have which vary from state to state.

ProDOS 8 is a copyrighted program of Apple Computer, Inc. licensed to Kitchen Sink Software, Inc. to distribute for use only in combination with *MicroDot*. Apple Software shall not be copied onto another diskette (except for archival purposes) or into memory unless as part of the execution of *MicroDot*. When *MicroDot* has completed execution Apple Software shall not be used by any other program.

WELCOME TO MICRODOT!

Welcome to *MicroDot*, the compact programming environment for the Apple II computer. Before we dive in to *MicroDot* and explain how everything works, we'd like to take the time to make clear to you what *MicroDot* is. This is a difficult task because the only other program that's anything like it is BASIC.SYSTEM.

Many people don't realize it, but BASIC.SYSTEM is not part of ProDOS. ProDOS does not understand BASIC commands at all. ProDOS is a disk operating system and as such it deals primarily with disk drives, but it also handles interrupts, memory allocation, and processes commands given through its MLI, the machine language interface. ProDOS does not collect garbage, manage Applesoft variable and program storage, or process commands such as LOAD THIS or PRINT CHR$(4);"OPEN THAT." The only commands ProDOS understands are numbers -- a pretty unfriendly way to communicate with a computer. Parsing BASIC commands is the job of BASIC.SYSTEM, and admittedly, it's pretty good at it. In short, then, BASIC.SYSTEM is an interface between Applesoft BASIC and the cold, hard reality of ProDOS. BASIC.SYSTEM takes what you type and converts it to a series of numbers that ProDOS can understand. In addition, BASIC.SYSTEM also handles garbage collection, BASIC memory allocation, and a variety of other things that ProDOS doesn't.

BASIC.SYSTEM is a program whose main job is to convert what you type (for example, LOAD NUMBERS) to the machine-language commands that ProDOS understands. These types of programs are called shells. BASIC.SYSTEM is unique in that it allows you to program in BASIC and use these commands in your programs. Well, it was unique, until now.

MicroDot is a replacement for BASIC.SYSTEM. Like BASIC.SYSTEM, its primary job is to facilitate communication between BASIC and ProDOS. *MicroDot*, like BASIC.SYSTEM, is (in computer jargon) a ProDOS command shell for BASIC programmers, or, if you like, a programming environment. Unlike BASIC.SYSTEM, which processes commands by watching the output streams for a control-D character, *MicroDot* communicates

directly with Applesoft. When you send it a command, it uses Applesoft subroutines to figure out which command you are talking about and to determine the parameters of each command, instead of having to watch what you put in a PRINT statement. Thus *MicroDot* requires much less code to do BASIC.SYSTEM's primary job: comprehend your commands.

Instead of supporting a collection of powerful commands with many options, as does BASIC.SYSTEM, *MicroDot* has a set of simple commands which can be used together to perform tasks. It's a building-block approach. You can put the commands together in almost infinite combinations. *MicroDot* has twenty-six most-used commands built-in; others can be installed if they are needed.

MicroDot gives you almost direct access to ProDOS, instead of insulating you from it as BASIC.SYSTEM does. You have almost limitless power over your disks and files. You'll feel like you're inside your programs, instead of on the outside looking in. And because of the ideas behind *MicroDot*, it's over 6K shorter than BASIC.SYSTEM.

MicroDot has built-in commands for loading and saving BASIC and binary (machine language) programs and files, for manipulating files (create, rename, delete), for handling pathnames and prefixes, for reading and writing sequential text files, and for reading directories.

You can also install additional commands (called modules) when they are needed to display a directory listing, to handle multiple open files at once, to randomly access text and binary files, to directly call the MLI, and to pack and unpack hi-res pictures directly to and from disk. In addition you will find a fast disk formatter, a simple BASIC line editor and a Program Writer patch.

This manual describes *MicroDot*'s commands and options. There's a lot of power at its heart. There are also some example programs on the *MicroDot* disk which you can load, print out, and study in conjunction with the text.

MICRODOT COMMANDS

ISSUING COMMANDS

MicroDot is ampersand-driven; therefore, all commands start with an ampersand (&). *MicroDot* has been designed so that you can use other ampersand routines with it, so you don't actually lose the use of the ampersand. See the section entitled "Ampersand Compatibility" (p. 49) for details. You can, instead, use CALLs if you want. See Global Location ALTENT.

After the ampersand comes the *MicroDot* attention character, the period (also known as a dot, hence the name *MicroDot*). This tells *MicroDot* that it should decode and execute this command. (If there is no period, *MicroDot* passes the command on to any other ampersand routines which are installed.) After the period comes the actual command, which is one or two letters in length, and is followed by any parameters needed by the command, such as pathname, address, and so on.

Commands may be issued from immediate mode or in a program. In both cases the format of the command is exactly the same: first an ampersand, then a period, next the name of the command, and finally, any parameters. You may also put multiple commands on a line, or mix *MicroDot* commands with regular Applesoft commands on the same line. Commands are issued the same way regardless of whether they are built-in *MicroDot* commands or optional modules loaded from disk.

If you are issuing *MicroDot* commands from immediate mode, you should not enter lines longer than 128 characters. This is because *MicroDot* uses part of the keyboard buffer for parsing commands.

COMMAND SYNTAX

In the command syntax descriptions on the following pages, a few special terms and symbols are used to describe parameters. These should not be typed as part of the command; instead, substitute an appropriate variable or expression.

var A numeric variable; may be either integer or real.

var$ A string variable.

exp A numeric expression; can be a constant, a function, a variable, or a combination using arithmetic symbols. In short, any legal Applesoft numeric expression, with a value of 0-65535.

exp$ A string expression; can be a literal (enclosed in quotes), a string function, or a concatenation. In short, any legal Applesoft string expression up to 63 characters in length.

[ ] Brackets around a parameter indicate that it is optional. If it is not supplied, a default value will be used, or the function of the command will be changed in some way.

RESIDENT COMMANDS

MicroDot has twenty-six built-in commands. These are all described in detail in the pages that follow, in alphabetical order.

&.A

(Append Data To File)

Syntax: &.A
Example: &.A

The &.A command moves the position-in-file pointer to the end of a file. This is most useful when adding new information to the end of a sequential data file. The next data written to the file will be added to the end of the file instead of overwriting existing data. The command requires that a file be open (see &.O). The command does not enter write mode after moving to the end of the file (see &.W).

&.BL

(Load a Binary File Into Memory)

Syntax: &.BL,exp$[,exp]
Examples: &.BL,"PICTURE",8192
&.BL,"ML.PROGRAM"

The &.BL command is very similar to the BASIC.SYSTEM BLOAD command. Like BLOAD, &.BL loads data from a binary (type BIN) file on the disk to an address in memory. The first parameter is the pathname of the file to load. The second parameter is optional; it is the address at which to load the file. If a load address is not specified, the file is loaded at the address from which it was saved. The file must be a BIN file. If it is not, *MicroDot* will issue an error. Unlike BASIC.SYSTEM's BLOAD command, the &.BL command does not provide a way to load part of a file, or start reading somewhere other than the beginning of the file, or load files other than BIN files. To do this, you can use the RANDOM module discussed later in this manual.

&.BO

(Overlay a BASIC Program)

Syntax: &.BO,exp$,[#]exp
Examples: &.BO,"MODULE.1",#10000
&.BO,"ACCESSORY.1",16385

The &.BO command is a powerful command that allows great flexibility in writing Applesoft programs. It provides an easy way to swap parts ("modules") of your BASIC program in and out from disk, which means that you can write programs which are too large to fit into memory at once. The first parameter is the pathname of the module to load. It must be a BAS file. The second parameter can be either a line number or an absolute address, depending on the presence or absence of the number symbol.

If a number symbol is present, as in the first example above, *MicroDot* loads the module on top of the specified line. This means that all lines from 10000 through the end of the program will be deleted, and the module will replace the deleted lines. All variables will remain intact unless the module overwrote them inadvertently. If a number sign is not present, as in the second example, *MicroDot* loads the module at the specified absolute address. This can be used for loading half of a program below the hi-res pages, while loading the other half above. (The first half would use &.BO to load the second half.) It can also be used to load an independent subprogram (like a desk accessory) at a certain address. By changing Applesoft's start-of-program pointer at 103-

104, the "desk accessory" can be run and then return to the main program.

If you use &.BO with an absolute address (i.e., no number sign), make sure that the memory address before the loading address contains a zero. This is an Applesoft requirement. In the second example, you might POKE 16384,0 before issuing the &.BO command. The &.BO command is like a cross between the &.L (Load) command and the &.BL (BLOAD) command. Like &.BL, it does not adjust the Applesoft pointers; like &.L, it relocates the program so that it will work correctly at the new address. &.BO does not turn off the current ONERR GOTO; you should make sure that ONERR GOTO still makes sense with a different module in memory.

See the chapter "Programming With Overlays" for more detailed instructions on performing overlays.

&.BS

(Save Binary File From Memory)

Syntax: &.BS,exp$,exp,exp
Example: &.BS,"PIC.1",8192,8192

The &.BS command is like BASIC.SYSTEM's BSAVE command. It creates a BIN file and saves an image of the memory range specified into this file. (You may also use an existing file, as long as it is of type BIN.) The first parameter is the pathname of the file. The second parameter is the memory address at which to start, and the third parameter is the number of bytes (length) to save. Both parameters are required, even if saving to an existing file. If you are saving a range of memory to an existing file, and the new data is shorter than the existing data, the file will be truncated (cut off) at the end of the new data. The address specified will be put into the file's AUXTYPE.

The &.BS command is much like the &.BL command in that you cannot use it to perform random-access writes on a file. You can use the RANDOM module for this.

&.BX

(Execute a Binary File)

Syntax: &.BX,exp$[,exp]
Examples: &.BX,"ML.PROGRAM"
&.BX,"ML.PROGRAM",768

&.BX is analogous to BASIC.SYSTEM's BRUN command. It loads a BIN file into memory and then jumps to its starting address. The syntax and parameters of the &.BX command are identical to those of the &.BL command.

Remember, not every BIN file is an executable program. Attempting to execute a non-executable BIN file can cause your computer to hang, crash, or have a fit.

&.C

(Close an Open File)

Syntax: &.C
Example: &.C

&.C, like the BASIC.SYSTEM CLOSE command, closes the currently open data file. (Unless you are using the MULTI module, only one file may be open at a time.) Any data which has been written to the file but has not been updated to the disk will be saved to disk at this time. If no files are open, no error will occur. This means that it is safe for you to issue &.C just to make sure the file is closed.

You should always close your file before your program ends. *MicroDot* will not warn you if you forget to close the file, but you may lose some data, and there is a chance that your disk might be damaged. See the discussion of the &.O (Open) command for more information about data files.

To close all open files, issue: POKE 46047,0 : POKE 49044,0 : &.C

&.D

(Delete a file)

Syntax: &.D,exp$
Example: &.D,"JUNK"

&.D is the DELETE command. The specified file will be removed from the disk. &.D will not allow you to delete a locked file, or a subdirectory which contains other files.

&.F
(Find File & Verify Type)

Syntax: &.F,exp$[,exp]
Examples: &.F,"DATAFILE"
&.F,"TEXTFILE",4

The &.F command checks for the existence of a file. (The first parameter is the file's pathname.) If the file does not exist, an error will be generated. This usage is similar to BASIC.SYSTEM's VERIFY command, which simply checks to see if a file exists. If the second parameter (the filetype) is not used, *MicroDot* does not check the filetype of the specified file. If, however, a filetype is specified, *MicroDot* will check the file's actual type to make sure it matches the desired type. If the filetypes do not match, a file type mismatch error will occur.

Thus, the first example simply checks to make sure that DATAFILE is on the disk. It does not matter what type the file is. The second example makes sure that TEXTFILE exists, and, furthermore, that it is of type 4. (Type 4 is a TXT file. See "ProDOS Filetypes".)

&.G
(Get Directory Entry)

Syntax: &.G[,var$,var,var]
Examples: &.G
&.G,NAME$,TYPE,AUX
&.G,NAME$,TYPE%,AUX%

MicroDot does not have a command for displaying a disk directory (unless you install the CAT40 or CAT80 modules). However, it does have commands for reading a directory from a BASIC program, so it's very simple to add custom directory listings to your programs. If &.G is used by itself, as in the first example, *MicroDot* sets up the directory and gets ready to read it. You should always use &.G by itself after opening (&.O) a directory for reading, and before trying to get an entry. If &.G is used with parameters, as in the second example, *MicroDot* reads the directory and passes a directory entry back to the variables specified.

In this example, the file's name is passed back to NAME$, its type to the variable TYPE, and its auxtype to AUX. The auxtype will consider the first auxtype byte as the low byte of the number and the second auxtype byte as the high byte of the number passed. The variables can be reals or integers or any combination.

The auxtype contains useful information about certain types of files. For BAS and BIN files, the auxtype contains the address from

which the file was saved. For TXT files, the auxtype contains the default record length. If you design your own data files, the auxtype can be used for whatever you like.

In addition to the information passed back by the &.G command, you can also use certain PEEKs to get more information about the files, or about the volume. See "Peeks and Pokes." The &.G command uses an algorithm which is compatible with the AppleShare network. When you reach the end of the directory file, an error 77 will be issued. When reading the /RAM disk on 128K machines, you will get an error #76 instead. Just one more off-the-wall way that /RAM is different from other ProDOS volumes.

&.IG
(Get File Info)

Syntax: &.IG,exp$,var,var
Example: &.IG,"BINFILE",TYPE,AUX

The &.IG command gets a file's attributes. They are passed back in exactly the same order as with the &.G command. The &.IG command gets the file type and auxtype for a given file. As with &.G, the numeric variables may be either integers or reals. See "Peeks and Pokes" to see how you can get more information with PEEKs. See &.G for auxtype format.

&.IS
(Set File Info)

Syntax: &.IS,exp$,exp,exp
Example: &.IS,"PICFILE",6,16384

&.IS allows you to change a file's type and auxtype. The first parameter, as usual, is the pathname of the file. The second parameter is the filetype (see "ProDOS Filetypes" for more about filetypes), and the third parameter is the auxtype. See &.G for auxtype format.

You should always do an &.IG before doing an &.IS, even if you are changing both of the values that &.IS changes. This is because &.IS can set more than just the filetype and the auxtype; with the use of POKEs, it can also set other file information such as the access (lock/unlock) status of the file and the last modification date. Doing an &.IG before the &.IS makes sure that you don't accidentally change something that you didn't mean to change. See "Peeks and Pokes" for information on changing other file attributes, such as the lock/unlock byte.

&.K
(Kill To End Of File)

Syntax: &.K
Example: &.K

The &.K command can be used whenever you are writing information into an existing data file. If the new data is shorter than the old data, the &.K command will "cut off" any data in the file beyond the current position-in-file pointer. This prevents you from wasting disk space with old data. To use &.K in this way, simply issue the command after writing new data to the file and before closing it. This eliminates the rather clunky standard BASIC.SYSTEM practice of deleting a file before opening it to write new data to it.

&.L
(Load BASIC Program)

Syntax: &.L,exp$[,exp]
Examples: &.L,"STARTUP"
&.L,"MAIN.MENU",8193

This command loads a BAS file into memory. (If the file is not a BAS file *MicroDot* will issue an error.) If the second parameter is not given, the program will be loaded at the current location specified by the Applesoft start of program pointer (103,104). If a second parameter is specified, the program will be loaded at that address and 103,104 will be changed to point to that location.

The lowest that a program should be loaded is 2049. This is also the default address when Applesoft is first started up. It is up to you to make sure that the memory location just below the program contains a zero. This is an Applesoft requirement. For example, you might type POKE 8192,0 before issuing the &.L command in the second example. &.L turns off any ONERR GOTO which may be in effect.

&.M
(Make a New File)

Syntax: &.M,exp$,exp
Examples: &.M,"SUBDIR",15
&.M,"TEXTFILE",4

The &.M command makes (creates) a file. If the file already exists, a duplicate filename error will occur. The second parameter is the desired filetype. The first example creates a subdirectory named "SUBDIR" (type 15); the second example creates a text file (type 4). See "ProDOS Filetypes" for a list of filetypes supported by ProDOS.

When a file is created, its access (lock/unlock) bits are set to $C3 (unlocked), and the current time is read from the ProDOS clock (if a clock is installed) to stamp the creation and modification date/time fields. The auxtype of the file is set to zero. The file is completely empty.

&.N
(Rename a File)

Syntax: &.N,exp$,exp$
Example: &.N,"OLDNAME","NEWNAME"

The &.N command renames a file. If complete pathnames are specified (such as /DISK/SUBVOL/OLDNAME), they must be the same up to the last filename in the path. (You could not rename /DISK/THIS1/OLDNAME to /DISK/OTHER/NEWNAME, for example, because it would involve moving the file to a different directory.) A file may not be renamed if it is locked.

&.O
(Open Data File)

Syntax: &.O,exp$
Examples: &.O,"TEXTFILE"
&.O,"SUBDIR"
&.O,"BINFILE"

The &.O command is used to open a data file for reading or writing. The file must already exist on the disk; &.O will NOT create files. (Use &.T or &.M to create files.) &.O does not care what type of file it uses; to check the file type, use &.F, &.T, or &.IG. After opening a directory file, use &.G to read entries. With any other type of file, use &.R and &.W to read or write data from the data file. (You cannot write to a directory.)

MicroDot is equipped for sequential file access only. To do random access processing, use the RANDOM module on the *MicroDot* system disk. Sequential access data file commands include &.O, &.R, &.W, &.Z, &.A, &.K, and &.C.

MicroDot normally can handle only one open file at a time, unless you are using the MULTI module. Opening a file with &.O while another file is still open will result in the first file being closed before the new file is opened. If the MULTI module is installed &.O will have an additional possible parameter. It is explained in the MULTI module section.

&.PG
(Get Prefix)

Syntax: &.PG[,var$]
Examples: &.PG
&.PG,PRE$

The &.PG command gets the current ProDOS prefix and returns it to the specified variable. The prefix is ProDOS's way of keeping track of which disk (or subdirectory) it is working with at a particular time. The &.PG command allows you to easily find out what directory ProDOS is defaulting to, much like the PREFIX command under BASIC.SYSTEM. If you do not specify a string variable, *MicroDot* will print the current prefix on the screen. This feature is useful mainly from immediate mode.

&.PS
(Set Prefix)

Syntax: &.PS,exp$
Example: &.PS,"/DISK/SUBDIR/"

&.PS changes ProDOS's default directory to the specified directory. If the disk or directory is not found, *MicroDot* will return an error. If you set the prefix to a null string, you will have to fully qualify all pathnames (i.e., all pathnames will need to start with a slash to tell ProDOS what disk to find them on). A null prefix means that ProDOS is <u>not</u> keeping track of a default directory at all and that you will need to specify the complete pathname of every file you access. In general it is a good idea to avoid null prefixes. Remember that ProDOS only understands prefixes, not slot/drive specifications. Use the &.V command to get the name of a particular disk volume.

&.Q
(Quit *MicroDot* System)

Syntax: &.Q[,exp$]
Examples: &.Q
&.Q,"APLWORKS.SYSTEM"

&.Q allows you to leave *MicroDot* and run another system program. Although we hope you'll never want to leave *MicroDot*, we are realistic enough to realize that some people will want to use other programs occasionally. Issuing &.Q by itself quits to the standard ProDOS quit prompt. (You know, the unfriendly ENTER PREFIX OF NEXT APPLICATION message.) If you have a program selector such as ProSEL, Squirt, Better Bye, or the Finder, it will run at this time to allow you to select another application. Any applications of importance that you write should allow the user to quit in this manner.

If you issue &.Q with a pathname, as in the second example, *MicroDot* will run the specified SYS program. (It must be a SYS program.) *MicroDot* can quit to very large system programs. BASIC.SYSTEM, on the other hand, cannot execute some system programs because they are too large. If an error occurs while trying to run a system program, a normal &.Q will be executed.

&.R
(Read from Data File)

Syntax: &.R
Example: &.R

&.R allows you to read from a data file. When you execute this command, *MicroDot* hooks itself up to the I/O hooks so that all GETs and INPUTs will come from the disk file. This (and during the &.W command) is the only time *MicroDot* hooks itself up to the I/O hooks. An out-of-data error will occur when the end of the file is reached. Since *MicroDot* hooks itself up to the I/O hooks during reads, you should not use the IN# or PR# commands while reading is in progress. These commands will disconnect the read mode partially or totally. Use PR# or IN# before entering read mode, or issue &.Z first and then issue the commands.

While you are reading from a file, you cannot print anything to the screen. *MicroDot* does this so that you'd don't see the "?" prompts generated by the INPUT statement and so that the characters which are read from the disk are invisible. If you want to print something, issue &.Z first.

You can turn off read mode in several ways. First, you can use the &.Z command. Second, you can activate write mode (&.W); this automatically turns off read mode first. Third, you can close the file (&.C). Finally, *MicroDot* automatically turns off read mode if you get a disk error or if you press control-reset. If you turn off read mode with &.Z (or by switching to write mode with &.W), you can re-activate read mode by issuing &.R again. The input characters will be read from the current position-in-file pointer.

After GETting a character from a file, its full 8-bit value can be obtained by PEEKing 1004. This allows you to read all possible byte values from any type of file.

&.S
(Save a BASIC Program)

Syntax: &.S,exp$
Example: &.S,"MY.PROGRAM"

This command saves the current BASIC program to disk in a BAS file. (If the file already exists, it must be a BAS file.) The auxtype of the file is set to the address from which the file was saved. (*MicroDot* does not use the auxtype when loading files; this feature was included for compatibility with BASIC.SYSTEM, which does use the auxtype to determine how to relocate the program.) The file can then be used as a stand alone program, or as a module with the &.BO command (see the description of the &.BO command).

&.T
(Test and Create File)

Syntax: &.T,exp$,exp
Example: &.T,"TEXTFILE",4

The &.T command checks for the existence of the specified file. If the file exists, the file type is checked to make sure that it matches the specified type. (If the type does not match, an error is generated.) If the file does not exist, it is created.

When used just before the &.O command, this command makes &.O act like the BASIC.SYSTEM OPEN command. (If the file does not exist, it is created, etc.) The &.F, &.M, and &.T commands give you tremendous flexibility in creating files.

&.V
(Get Volume Name)

Syntax: &.V,exp,exp[,var$]
Examples: &.V,6,1
&.V,6,1,VOL$

Since ProDOS doesn't understand slots and drives, only pathnames and prefixes, this command was included so that you can easily convert slot/drive numbers to volume names. This allows you to ask the user what device his disk is in, and then access data files on it. This way the user does not need to know the name of each and every one of his disks. (AppleWorks operates similarly.)

The first two parameters are the slot and drive, respectively, of the desired disk device. The final parameter is the string variable to which *MicroDot* will pass the volume's name. The volume name will be in a format like this: /DISK/ (with slashes before and after the name).

If you don't specify a string variable, *MicroDot* will print the volume name on the screen. This is useful mainly from immediate mode, while programming.

&.W
(Write to a Data File)

Syntax: &.W
Example: &.W

&.W allows you to write to a data file. When you execute this command, *MicroDot* hooks itself up to the I/O hooks so that all PRINTs go to the disk file. This (and during the &.R command) is the only time *MicroDot* hooks itself up to the I/O hooks. Since *MicroDot* hooks itself up to the I/O hooks during writes, you should not use the IN# or PR# commands while reading is in progress. These commands will disconnect the read mode partially or totally. Use PR# or IN# before entering read mode, or issue &.Z first and then issue the commands.

Naturally, while you are writing to a file, you cannot print anything to the screen. If you want to print something, issue &.Z first. You can, however, use GET or INPUT to get data from the user; the only drawback is that the characters typed will not be echoed to the screen, but will go to the disk file, as will any prompts. Once again, issue &.Z first.

You can turn off write mode in several ways. First, you can use the &.Z command. Second, you can activate read mode (&.R); this automatically turns off write mode first. Third, you can close the file (&.C). Finally, *MicroDot* automatically turns off read mode if you get a disk error or if you press control-reset.

If you turn off write mode with &.Z (or by switching to read mode with &.R), you can re-activate read mode by issuing &.W again. The output characters will be written at the current position-in-file pointer.

&.X
(Execute BASIC Program)

Syntax: &.X,exp$[,exp]
Examples: &.X,"STARTUP"
&.X,"MAIN",16385

The &.X command loads the specified BASIC program, clears ONERR GOTO and all variables, and then RUNs the program, just like the RUN command under BASIC.SYSTEM.

Like *MicroDot*'s &.L command, &.X allows you to specify a location in memory to load the program. If no address is specified, the program is loaded at the current address specified by the start-of-program pointer (103-104). Also like &.L, you must make sure that the location before the program contains a zero for Applesoft.

From immediate mode, you can still use the RUN command to run the program currently in memory.

&.Z
(Zap (Deactivate) Read/Write)

Syntax: &.Z
Example: &.Z

&.Z is used to turn off either read mode (activated with &.R) or write mode (activated with &.W). You need to do this before issuing a PR# or IN# (or a GET or INPUT from the keyboard, or a PRINT to the screen) since *MicroDot* hooks itself up to the I/O hooks in zero page during read and write mode.

It does not hurt to issue this command even if you're not in read or write mode. You can use it at any time to make sure that what you print isn't going to the disk and what you input isn't coming from it. It's an especially good idea to include the &.Z command in your error handling subroutine.

Not Quite the End

With the basic commands above, you have the control over ProDOS that you need for most of your programs. If you only need the basic commands there is no need to load the modules that follow. The basic commands only occupy a little over 3K of memory! You still have almost 43K of space for your BASIC program and variables! Read on though. You may need some of the following modules at times. In addition, you find information about the blank disk formatter and technical information on PEEKs and POKEs that add power to the basic commands above.

OPTIONAL MODULES

Introduction

MicroDot has a great deal of built-in power. Many types of programs can be written using only *MicroDot*'s built-in commands. Some applications, however, will need abilities which *MicroDot* doesn't have. *MicroDot* has a number of optional modules for performing these functions.

These optional modules are stored on disk in standard BIN files and are installed with the normal &.BX command. Once a module is installed, you can use its commands as if it were a part of *MicroDot*, with the usual &. syntax. If you have been paying close attention, you might realize that five letters are unused by *MicroDot* commands. These letters are E, H, J, U, and Y. The optional modules hook themselves up to these letters. These letters are pretty much "leftovers," so there is unfortunately not much correspondence between a module's letter and what the module does.

Module	Letter	Function
CAT40	**E**	Displays directory listings in 40 columns
CAT80	**E**	Displays directory listings in 80 columns
PACK	**H**	Hi-res packer; packs/unpacks directly to disk
RANDOM	**J**	Random-access file commands
MULTI	**U**	Use more than one file at once
SYSCALL	**Y**	Perform ProDOS MLI calls

To install a module, simply use the &.BX command. For example, to install CAT40, you would enter &.BX,"CAT40". A sign-on message will appear on the screen and the module will be ready for use. You can install modules from immediate mode or from a program.

If you try to install the same module more than once, you will get an error message. This error message does not stop a running program, however. (This means that you can safely put a group of &.BX commands at the beginning of a program to load the modules used by that program. If you then re-run the program, you will get error messages from trying to load the modules, but the program will continue to run.) You can install modules in any order, except

for the MULTI module, which is best installed last. (See the MULTI instructions for special memory management considerations.)

Modules relocate themselves to high memory, setting HIMEM below their code to protect themselves. This means that installing a module will trash your string variables. Furthermore, modules initially load at 8192 ($2000), which could, in some cases, be on top of your program. For these reasons, you might want to write a short program which does nothing but load the optional modules which your main program uses, and then ends by running your main program.

Module: Catalog Disk

Name of module to &.BX: **CAT40** or **CAT80**

&.E

(Catalog 40 or 80 column)

Syntax: &.E[,exp$]
Examples: &.E
&.E,"/APPLEWORKS/"

MicroDot doesn't have a built-in command for displaying the contents of a disk directory. (Remember, though, it does have a command that lets BASIC programs read the contents of a directory.) The programs CAT40 and CAT80 on the *MicroDot* disk are two optional command modules which allow you to get a directory listing of a disk with a 40 or 80 column display, respectively. The modules are most useful when programming, but can also come in handy anytime you need a quick-and-dirty directory listing in a program.

The two modules function identically except for the format of the information displayed on the screen. You cannot have both the CAT40 and CAT80 modules in memory at the same time.

To produce a directory listing, simply issue an &.E command. If you don't specify a directory (as in the first example above), the current prefix directory will be cataloged. If you do specify a directory (second example), that directory will be cataloged.

The display of the CAT80 module is very similar to the display of the BASIC.SYSTEM CATALOG command, except that a few

columns have been moved around and have different formats. First, the asterisk denoting a locked file is between the filename and the filetype, not to the left of the filename. Secondly, the AUXTYPE and END.FILE columns have been moved to the left of the date columns. And finally, the dates are shown in the form MM/DD/YY instead of BASIC.SYSTEM's DD-MMM-YY format.

The top of the file listing displays the name of the directory and the column headings; the bottom line shows the total number of blocks on the disk, the number used and the number that are free.

The CAT40 module's display is similar, except that it does not display the creation and modification dates of files.

Module: Hi-Res PACKER/UNPACKER

Name of module to &.BX: **PACK**

Introduction

The hi-res packer module allows you to pack and unpack hi-res pictures, which means that they take up less disk space. If you are developing a graphics-intensive program, the PACK module might come in quite handy.

Unlike other hi-res packers, *MicroDot*'s packer does not require a large memory buffer to hold the packed picture. Instead, the packer packs and unpacks pictures directly to and from the disk. Unfortunately, this means that the program is somewhat slower at loading and displaying pictures than other packers, but it does give you more usable memory.

The PACK module is a specially modified version of the packer used by Polarware (Penguin Software), also published in "Graphically Speaking" by Mark Pelczarski, and is used by permission. This also means that if you create a packed picture using a Polarware/Penguin product such as The Complete Graphics System, you should be able to unpack it with the PACK module.

&.HP
(Hi-res Pack)

Syntax: &.HP
Example: &.HP

This command packs a hi-res picture to disk. You must open a file before issuing this command, and you must close it afterward. This allows you to store more than one packed picture in a file (or both banks of a double Hi-Res picture).

&.HP packs the screen specified by location 230, an Applesoft pointer which is also used to determine the drawing page. If you have issued an HGR or HGR2 command this location is automatically set to the correct value for that hi-res page. You can also POKE 230 to tell &.HP which hi-res page to pack; POKE 230,32 for page 1, and POKE 230,64 for page 2.

&.HU
(Hi-res Unpack)

Syntax: &.HU
Example: &.HU

This command is the reverse of the &.HP command. Like &.HP, you must open and close the file yourself. The hi-res page used is determined by location 230.

Module: RANDOM ACCESS FILES

Name of module to &.BX: **RANDOM**

Introduction

MicroDot has, built-in, the ability to read and write sequential data files. For many programs, that's all you will need. If, however, you do need random-access file capabilities, the RANDOM module can provide them for you.

A random-access file is a file with fixed-length records. For example, each record might be defined as 100 bytes. This allows you to easily access any desired record; since each record is the same length, it is easy to move the position-in-file pointer to any

given record. The records do not need to be accessed sequentially; they can be accessed in any order.

Here is a brief overview of how to use the RANDOM module; command syntax is covered after the overview.

CREATING A NEW RANDOM-ACCESS FILE

Creating a new random-access file involves only a little more work than creating an ordinary sequential file. First, you use &.M to create the new file. Then, to maintain compatibility with BASIC.SYSTEM, you should store the file's record length in its auxtype. In fact, this will come in handy if you ever forget what the record length is; all you have to do is check the file's auxtype to find out.

To put the file's record length in its auxtype, first use the &.IG command to get the file's current attributes. Remember, you should always perform an &.IG before an &.IS to avoid inadvertently changing a file's attributes. Then issue an &.IS with the desired record length as the auxtype parameter. That's all there is to it.

OPENING AN EXISTING RANDOM ACCESS FILE

You can open a random-access file with the standard &.O command. As usual, you might want to include an &.F or &.IG before the &.O to make sure the file is of the correct type.

You might also want to get the file's default record length from its auxtype with the &.IG command.

ACCESSING THE RECORDS IN THE FILE

Before you can read or write a record, you must first move to that position in the file. The RANDOM module includes a command which allows you to move the position-in-file pointer to any given byte in the file. To access a particular record, simply multiply the record number times the record length to get the byte number. You may also add a byte offset to access a particular byte in the record. In terms of the R, L, and B parameters used with BASIC.SYSTEM's

OPEN and READ commands, the formula you need to know is R * L + B.

Once you have moved the position-in-file pointer to the appropriate location, you can use &.R and &.W to read or write data, just as with an ordinary sequential file.

ANOTHER POWERFUL RANDOM FEATURE

The RANDOM module also allows you to move groups of bytes en masse between the Apple memory and the disk. Coupled with the ability to move the position-in-file pointer to an arbitrary byte in the file, this feature allows you to do random-access memory reads and writes from any size file, much as the BLOAD and BSAVE commands of BASIC.SYSTEM do when used with the B parameter. You will notice that since *MicroDot* requires you to open the file only once, it is faster than BASIC.SYSTEM which requires you to open the file every time you issue a BLOAD or BSAVE.

&.JF
(Skip Fields)

Syntax: &.JF,exp
Example: &.JF,3

&.JF is similar to BASIC.SYSTEM's POSITION command, or to the F parameter on the BASIC.SYSTEM READ command. Starting at the current file position, &.JF skips over the specified number of carriage returns, moving the position-in-file pointer past the specified number of fields.

The maximum number of fields you can skip is 255. If you specify zero, no fields will be skipped.

This command can be used with sequential text files to skip to a certain line of the file, but its real use is with random-access text files to read a particular field of a record. If the file has a carriage return between each field of each record, you can use &.JP to position the file pointer to the beginning of the record, then use &.JF to position the pointer to a specific field within that record.

&.JG
(Get File Pointer)

Syntax: &.JG,var
Example: &.JG,PTR

The &.JG command tells you, in bytes, exactly where the file pointer is right now. This allows you to, among other things, move the pointer forward or backward from the current position. For example, you might do an &.JG,PTR then do an &.JP,PTR-1 to move back one byte in a file. There are a number of other uses for this command as well.

The variable you specify must be a real variable. A file may be up to sixteen megabytes in length; an integer variable is too small to hold a number that large.

&.JP
(Position File Pointer)

Syntax: &.JP,exp
Examples: &.JP,10000
&.JP,10 * L

The &.JP command allows you to move the position-in-file pointer to any arbitrary byte within the file. ProDOS files may be up to sixteen megabytes in length, and the &.JP command can handle files that large. The first byte of a file is byte zero, and the maximum byte number is 16,777,215. If you specify a byte that is beyond the end of the file, the file will automatically be extended.

The &.JP command works with bytes, not records. To convert a record number to a byte number, simply multiply the record number by the record length, as in the second example above. (Remember, most *MicroDot* parameters can be expressions.) The first record of a file is record zero, not one. You can also add a byte offset to this calculation. Note that this means you can start reading or writing at any byte within any record.

This command works on any open file. ProDOS and *MicroDot* don't care if the file is TXT or not, or if it was originally created as a sequential or random-access file.

If you use record numbers to access a file, it is up to you to keep track of the appropriate record lengths. This isn't so bad when you're working with just one file, but if you're using the MULTI

module in conjunction with the RANDOM module, you might want to use an array to hold all the information about each file. At the very least, use a variable to hold the record length. Then, if you decide to change the record length during programming, you need only change the value for the variable, not dozens of lines of code.

&.JR
(Read From File)

Syntax: &.JR,exp,exp
Examples: &.JR,AD,LN
&.JR,8192,8192

The &.JR command is very similar to the &.BL command, in that it reads bytes from a file on disk directly into memory. However, unlike &.BL, &.JR does not open the file nor does it close the file when done reading, and it doesn't require a BIN file. You must open the file yourself before issuing &.JR, you must close it yourself afterward, and you must check the file's type yourself (if you wish) with the &.IG or &.F commands. This makes the &.JR command a flexible tool.

When used with the &.JP command, &.JR can be used to emulate the abilities of BASIC.SYSTEM's BLOAD command, i.e., randomly accessing any portion of a large file and loading that portion into memory. You might use this command to read through a sequence of hi-res pictures stored in one file, or to load first one half of a double-hi-res picture followed by the second half.

If you attempt to read more data than is left in the file, you will not get an error. This is not a bug; BASIC.SYSTEM works the same way. You can check the length of the last BLOAD (see "Peeks and Pokes") to make sure that you got all the data you asked for.

&.JW
(Write memory to File)

Syntax: &.JW,exp,exp
Examples: &.JW,AD,LN
&.JW,8192,8192

The &.JW command is to the &.BS command as &.JR is to &.BL. It requires that the file be open. The file may be of any type. You are responsible for opening and closing the file yourself. See the discussion of the &.JR command for more information.

Module: Multiple Files

Name of module to &.BX: **MULTI**

Introduction

The MULTI module is used when you need more than one open file at once. Normally *MicroDot* only allows one file open at once, but the MULTI module can handle up to 8 files.

You open and close files with MULTI in exactly the same way you do normally, with the &.O and &.C commands. With MULTI installed, these two commands do a lot more than open and close files, but that doesn't matter, since the end result is the same.

All existing programs will work fine with MULTI installed, as long as they use standard *MicroDot* commands (or ML entry points) to access files.

Some method is needed to tell the MULTI module which of the open files you wish to use at a given time. For example, if you have five files open, you need to be able to tell MULTI to switch to #2, then #4, etc. This is accomplished by using the reference numbers which ProDOS assigns to files when they are opened. These numbers are assigned starting with number 1 and go up to 8.

Unless you issue a global close at the beginning of your program, though, you cannot assume that the first file you open will be reference number 1. The previous program might have left a file open, or an exec file could be running. For this reason, you should always either issue a global close at the beginning of your programs, or use a special enhancement of the &.O command, explained next.

Enhancement of &.O command from standard *MicroDot* commands

&.O Syntax: &.O,exp
(Open a data file) Example: &.O,RN

When MULTI is installed, the &.O command has a facility to return the reference number of the file just opened to a variable. Simply place a comma and a numeric variable after the pathname in the &.O command: &.O,"ACCUDRAW",RN. This example will put the file's reference number into the variable RN. Naturally you are free to use any legal real or integer variable name. You will use this number to select which file to work with at a given time.

IMPORTANT NOTE

You should not install other modules while more than one file buffer is available. It will seem to work, but if you then change the number of buffers, you will de-allocate the memory used for the newly installed modules, and the system will crash as these modules are overwritten by file data and/or strings.

The solution is simple. You can either make sure that MULTI is always the last module installed. Or, alternately, you can close all files and then issue an &.UB,1 command to de-allocate all buffers, then install any other desired modules, and then re-allocate the buffers with the &.UB command.

BUFFER ALLOCATION

Buffers are allocated in the following order: first, the standard *MicroDot* buffer at $BB00 is allocated. Then the buffer immediately below *MicroDot* (usually at $AD00, if MULTI is the only module installed) is allocated, and so on, using buffers further and further down in memory. Note that unlike BASIC.SYSTEM, *MicroDot* does NOT move your string variables down when opening files. Instead, you tell it how many files you will be using, and it reserves enough memory for that many buffers.

If you have only one buffer allocated (&.UB,1), the standard *MicroDot* policy of closing the current file when another one is opened remains in effect. When you have allocated more than one buffer, MULTI will not know which file to close to make room for the new one. In this case, if you try to open more files than there are buffers allocated, you will get an error.

&.UB Syntax: &.UB,exp
(Set Number of Buffers) Example: &.UB,3

Use the &.UB command to tell *MicroDot* how many files this program will be using. The MULTI module will allocate the desired amount of memory by lowering HIMEM. This means that you should always use &.UB before defining any string variables. If you remember DOS 3.3 at all you should recognize this command as being similar to MAXFILES.

Be sure to use &.UB,1 at the end of a program which uses more than one file. (After closing all the files, of course.) This will release the memory used by that program's file buffers and make it available to Applesoft again.

Attempting to use the &.UB command while files are open will result in an error.

&.US Syntax: &.US[,exp]
(Select File to Use) Examples: &.US,1
&.US

This command tells *MicroDot* that you want all file input and output to use the specified file. You must specify the file's reference number, as passed back by the &.O command. The &.R, &.W, &.K, &.A, and all of the RANDOM commands (if you have RANDOM installed) will use the specified file. When you open a file using &.O, that file is automatically selected for use.

Module: SYSCALL

Name of module to &.BX: **SYSCALL**

Introduction

The SYSCALL module is the simplest of the *MicroDot* modules (also the smallest in terms of bytes used), but it's one of the most powerful, as well. SYSCALL allows you to directly access the ProDOS MLI from BASIC.

We strongly suggest getting a ProDOS reference book such as Apple's manual or Quality Software's "Beneath Apple ProDOS" if you intend to use the SYSCALL module. It is definitely not for people who don't know what they're doing.

&.Y

(System Call To ProDOS MLI)

Syntax: &.Y,exp,exp[,var]
Examples: &.Y,130,0
&.Y,128,768,ERR

The first parameter of the &.Y command is the MLI call number. Remember that this number must be specified in decimal, not in hexadecimal as it is described in most ProDOS references.

The next parameter is the address of the parmlist for this MLI call. This parmlist can be set up via POKEs. It is beyond the scope of this manual to describe the formats of the various parmlists; check your ProDOS reference.

The final, optional parameter is a variable which, if present, will contain the error code resulting from the MLI call, or zero if no error occurred. If the final parameter is present, the error code will be passed back to the variable; if the final parameter is not present, any errors will be passed back through *MicroDot*'s standard error handling routines and can be trapped with ONERR GOTO.

This feature allows you to check for an error without having to use ONERR GOTO. Simply put a statement like IF ERR THEN... after an &.Y command to check for errors.

By the way, the first example is a ProDOS GET_TIME call, which updates the ProDOS global page date/time locations with the current date and time (if a clock card is installed). You do not need to set up a parameter list for this call.

The second example is a READ_BLOCK call, using the parmlist at 768. See your ProDOS reference for more information on READ_BLOCK and the format of the parameter list.

SYSCALL can be very useful for performing AppleShare network calls. Unfortunately, information on these calls was not available at this writing. Contact Apple Technical Support for further information.

ProDOS CLOCK FIX PATCH

If you have a clock in your computer, you may have a few problems issuing *MicroDot* commands from immediate mode. You may get a ?SYNTAX ERROR message for a syntactically correct command, or the command may execute but give you a ?SYNTAX ERROR afterward. This is a problem which is caused by your computer's clock; this bug can be fixed with the CLOCK.FIX module.

The official ProDOS clock protocol allocates memory locations $200-$27F (512-639) for use by the clock card. ProDOS calls the clock driver whenever a MLI command such as OPEN, READ, etc. is issued. In other words, whenever the disk is accessed (or a ProDOS call of any type is performed), $200-$27F may get trashed. Unfortunately, this is where Applesoft stores your immediate mode commands, so if you have a clock in your system, it is possible that ProDOS will turn your command into garbage even as *MicroDot* attempts to execute it.

The CLOCK.FIX module is the solution. It hooks itself up to ProDOS's clock routines so that whenever ProDOS needs the current time and date, it calls CLOCK.FIX instead of the regular clock driver. CLOCK.FIX saves the contents of $200-$27F in a safe location, calls the clock driver, and then restores the saved bytes back to $200-$27F. The result is that there is no net change in the

contents of the keyboard buffer, so immediate mode commands work correctly.

CLOCK.FIX is only needed when you are developing programs and are issuing commands from immediate mode. *MicroDot* commands always work fine in programs even if you have a clock installed.

The STARTUP program on the *MicroDot* system disk automatically installs CLOCK.FIX when you boot the disk. (If your computer doesn't have a clock, CLOCK.FIX will print "NO CLOCK IN SYSTEM" and will not install itself.) Some clocks (such as the one built into the IIgs) do not use $200-$27F and therefore there is no conflict with *MicroDot*; you won't need the CLOCK.FIX module for those clocks.

CLOCK.FIX also hooks itself up to the ProDOS quit routines to de-install itself when you quit *MicroDot*. If it didn't do this, CLOCK.FIX would quickly be overwritten by other routines when the next SYS program was executed and your system would then crash or lock up when ProDOS tried to get the current date and time.

PROGRAMMING TIPS

MicroDot EXEC FILES

Introduction

MicroDot does not directly support a command similar to BASIC.SYSTEM's EXEC command. It is, however, possible to perform a similar function with the &.O and &.R commands, since all *MicroDot* commands can be issued from immediate mode.

The basic idea of an exec file is that instead of input coming from the keyboard, as usually happens, all input comes from a text file on the disk. Everything in the disk file is accepted as input as if you had typed it yourself. This includes Applesoft immediate mode commands, Applesoft program lines, *MicroDot* commands, and Monitor commands.

You might have already guessed how to get *MicroDot* to do this, based on the hints given in the first paragraph of this section. Simply open the exec file with &.O, from immediate mode, and then issue &.R, also from immediate mode. You can even do this as a one-line command, by separating the two commands with a colon, like this:

```
&.O,"EXECFILE" : &.R
```

Exec files created with BASIC.SYSTEM can be executed this way, except of course if they contain BASIC.SYSTEM commands such as LOAD, SAVE, etc. Applesoft program listings typically exec fine. When the end of the file is reached, you will see ERR 76, and the Applesoft prompt will re-appear. *MicroDot* does **not** automatically close EXEC files, so you should close it yourself with &.C.

When execing a file, *MicroDot* does not echo anything to the screen, not even the Applesoft bracket prompts. If a PRINT statement is executed, for example, it won't appear on the screen. Nothing will be printed until the end of the file is reached, and the error message

is printed. Don't worry, there are ways to print things on the screen from within an exec file (You'll read how in a few paragraphs).

CAPTURING APPLESOFT LISTINGS IN TEXT FILES

A procedure closely related to exec files is the technique of putting Applesoft program listings into text files. With BASIC.SYSTEM, this required adding a line to your program and running it. With *MicroDot*, it can be done from immediate mode. All you need to do is issue a command like this:

```
&.M,"CAPTURE",4 : &.O,"CAPTURE" : &.W : LIST : &.C
```

This creates the capture file, then opens it, sets *MicroDot* up to write to the file, and issues a LIST command. The listing will be printed to the file instead of the screen. Then the file is closed by the &.C command.

Coupled with the exec file technique described above, this trick allows you to edit your Applesoft programs using a word processor.

WRITING EXEC FILES FOR *MICRODOT*

The above examples make *MicroDot*'s exec file capability seem a little limited, especially the fact that nothing can be printed to the screen. However, *MicroDot* is really quite flexible. Certain *MicroDot* commands can, when included in an exec file, give it extra flexibility. We are now talking, of course, about going beyond simple program listings and into more powerful and imaginative uses of exec files.

Avoiding error statements at the end of the file

The first refinement to *MicroDot*'s exec file capability would be to include an &.C command at the end of the exec file. This way, the file closes itself before you get the ERR 76 message.

Making PRINT statements appear on the screen

To cause the EXEC file to echo on the screen, include the following immediate mode line in your exec file:

```
POKE 54,PEEK(996) : POKE 55,PEEK(997)
```

All input will still come from the exec file, except that now everything will be echoed to the screen (or whatever output device was active when you executed the file). These POKEs restore the output hooks to what they were when the &.R command was issued. To go back to no-echo mode somewhere in your exec file, simply include another &.R command. DOS 3.3 users will recognize this trick as the equivalent of MON O,I and NOMON O,I. You can also use it when debugging to see what is really in those text files you're reading, or to write a "read and show text file" utility.

Allowing keyboard input, etc. in the middle of an EXEC file

The &.Z command can also be useful. Consider the following line:

```
&.Z : PRINT "HI THERE!" : &.R
```

When this is executed, the &.Z command will turn off exec mode. Then the message HI THERE! will be printed on the screen. Then exec mode will be re-enabled, and off we go again. You could also use a line like the following:

```
&.Z : PRINT "TYPE &.R TO CONTINUE"
```

When this is executed, the message will be printed on the screen and the Applesoft prompt will appear. At this point, you can issue Applesoft commands or do whatever else you would like. Then you can issue &.R and the exec file will take off again right where it left off. If you want to permanently quit the exec file, type &.C when the prompt comes back.

RANDOM module tricks

With the RANDOM module installed, you can include &.JP,0 to cause an exec file to start over again at the beginning. You can also use &.JF to skip over the next lines in the exec file (most useful with an IF statement in the exec file).

Several quick tricks

You can use the WAIT -16384,128 : POKE-16368,0 sequence to wait for a keypress within an exec file.

You can have an exec file type in a little program which turns off exec mode with &.Z, runs, and then turns exec mode back on with &.R.

But... You canNOT do this

Unfortunately, you can't issue any *MicroDot* commands which open a file (This includes &.L, &.S, &.BL, and &.BS) from within an exec file because *MicroDot* only allows ONE open file at a time UNLESS you install the MULTI module. (Well, you can use these commands, but they will close the exec file to open the other file.) If the MULTI module is installed, you CAN open files from within an exec file. Or... you can have one exec file chain to another one by closing the current file, opening the new file, and issuing an &.R (all on the same command line, of course).

Not the end yet!

I hope that this has demonstrated to you some of the interesting things you can do with exec files with *MicroDot*. *MicroDot* is as flexible with exec files as it is with everything else; the possibilities are wide open. By the way, *MicroDot* can automatically exec a file on startup; just make sure that the text file to exec is named STARTUP on the disk.

PROGRAMMING WITH OVERLAYS

Introduction

MicroDot's &.BO command is a powerful tool for creating programs that are larger than your Apple's memory. With it, you can break your program up into overlays (modules) which are loaded from disk when needed. &.BO does away with the clumsy binary-file overlays that you may remember from the DOS 3.3 days. (It's still used some under ProDOS, although not as much, because BASIC.SYSTEM has a CHAIN command that works.)

Here, in a nutshell, is what the overlay technique is all about. Typically, you have a main menu for your program which stays in memory at all times (the "core"). In addition, you also might have some subroutines which are used throughout the program. In a normal program, you might then have a several subroutines which are activated from the main menu; in a program that uses the overlay technique, only one such subroutine is resident in memory at a time. The main menu and shared subroutines remain in memory at all times; the routines which are called from the main menu are called into memory only when needed.

HOW TO MAKE OVERLAYS

With *MicroDot*, making overlays is very easy, because you can make modifications to the core of the program without fear of messing up your overlays. Under DOS 3.3 making overlays required you to compute the location where the overlay was to occur. The overlay modules had to be BSAVEd from that location. If the core part of the program was modified, you had to re-compute your overlay location and then re-do all of the BSAVEd modules. With *MicroDot*, all of this is automatic. And you don't save your modules as binary files, you save them as BASIC files. You don't even have to have the core section of the program in memory as you write the overlay (until you are ready to start testing).

If you think about the above paragraph you will soon realize that once you write a BASIC module you can use it with ANY BASIC program you write in the future. Now imagine that you save all of

your favoite subroutines as small modules. You can load those modules into any program at any time. This could significantly reduce your programming time once you have "cropped" and saved your favorite routines out of old programs. **WARNING: The &.BO command does NOT change the End of Applesoft program pointer (175, 176)! You would have to do that manually if you use this technique.** You could load it into an overlay area without worrying about the End of Applesoft pointer. Simply renumber your routines to high line numbers. After loading (&.BO) onto your new program, renumber the module to place it where you want it in your new program. Program Writer makes this operation a piece of cake. Hmmm... I feel a "Quality BASIC Subroutines" disk coming on...

Here are some tips for implementing the overlay technique using *MicroDot*'s &.BO command. First, the overlays should be standard BAS type Applesoft program files. The overlays should all use line numbers higher than the highest line of the main program. For example, the main program might have line numbers less than 20000, and the overlays would have line numbers greater than or equal to 20000. The overlays may all have the same line numbers if you like (in fact, this makes the main menu routine easier to write; just load the appropriate overlay and GOTO or GOSUB 20000).

You must use the LOMEM command at the beginning of the main program to make sure that your variables are out of the way of your overlays. Simply check the PRGEND pointer ($AF-$B0 or 175-176) for the end of the main program and then add to that address the length of the longest overlay. You can even do this from within your main program so that you don't have to change the LOMEM statement each time you modify your program. To do this, get the address of the end of the program into a variable, use &.IG and a couple of PEEKs (see "PEEKs and POKEs") to get the length of the longest module.

You must take string literals into account when writing your overlays. A string literal is a statement such as A$ = "HELLO". This kind of string is not moved to the string pool just below HIMEM; the string's descriptor points instead to a location within the program. If you define a string like this in one of your overlays, and then load a different overlay, the string will become garbage. If you want

strings like this to be permanent, force Applesoft to move them to the string pool with a statement like A$ = "HELLO" + "".

There's one last thing you should be aware of, and that's error trapping. Applesoft's ONERR GOTO statement works in a pretty weird fashion. When you're getting ready to load an overlay, you should make sure that if there is an ONERR GOTO in effect, the ONERR GOTO statement is not within the overlay which is about to be replaced. The ONERR GOTO statement also should not point to a line in the overlay. These considerations are only necessary when you're about to load a new overlay; otherwise, you can use ONERR as usual.

You should of course use the # parameter of the &.BO command to load your overlays on top of a specific line number, such as 20000 in our example. This will delete all lines after 20000 and replace them with the overlay from the disk.

Follow these guidelines and you'll find that the overlay technique is a powerful way to write large programs without overflowing your computer's memory.

Programs on the Disk

The files are on the *MicroDot* disk are so numerous that when you catalog the disk, about half of the files will scroll off the screen unless you're careful and hit Control-S quickly.

The listing below is a reproduction of the *MicroDot* directory listing, except that we've added comments for each file instead of the standard directory information. All of the BASIC programs on the disk can be loaded into memory with &.L and listed with LIST. The example programs cover many of the finer points of programming with *MicroDot* and are well-commented. Print them out and study them; you'll learn a lot. The files with lots of periods after their names (type $00 or NUL when in the directory of the disk) are simply dividers that separate the directory listing into sections. They serve no purpose other than decoration.

/MICRODOT/

PRODOS .....ProDOS 8 Version 1.6.
MICRODOT.SYSTEM....The *MicroDot* shell.
STARTUP.....BASIC program which loads CLOCK.FIX and CAT40 or CAT80 upon boothing.
MODULES .....(Optional add-on *MicroDot* commands.)
CAT40 .....40-column directory listing module.
CAT80 .....80-column directory listing module.
MULTI.....Buffer management module; allows up to 8 files to be open at once.
RANDOM.....Allows random-accessing of data files.
PACK.....Direct-to-disk hi-res packer/unpacker.
SYSCALL .....Performs ProDOS MLI calls from BASIC.
UTILITIES.....(Other programs that work with *MicroDot*)
EDIT .....Simple page-3 resident line editor.
PW.ADAPTER .....Adapt Software Touch's Program Writer for use with *MicroDot*.
FORMATTER .....BASIC program to format disks.
FORMAT.1 .....Uses $2000-$3FFF (Hi-res page 1)
FORMAT.2 .....Uses $4000-$5FFF (Hi-res page 2)
FORMAT.3 .....Uses $6000-$7FFF (Hi-res page 3)

EXAMPLES .....(List and learn)
LOAD.MODULES .....One way to load modules from your own BASIC programs.
DATE.TIME .....Prints the date and time of the last ProDOS call (usually disk access).
READ.DIR.....Demonstrates how to read a directory from BASIC.
LOCK.UNLOCK .....How to lock and unlock files.
OPEN.FILES.....How to open new and existing files.
FILE.TYPES .....A brute-force routine to convert file type numbers to three letter codes.
ERROR.CODES.....Another brute force routine; converts ProDOS/*MicroDot* error codes to English.
LAST.DISK.....Displays the slot, drive, and name of the last disk accessed.
LIST.DISKS.....Lists all online volumes by slot, drive, and name.
PACK.UNPACK.....BASIC subroutines demonstrating use of the PACK module.
FLUSH.....How to emulate BASIC.SYSTEM's FLUSH.
CHANGE.STARTUP.....Change the startup program run by *MicroDot* (and other programs) on boot.
DISK.SCAN.....How to scan a disk for bad blocks using the SYSCALL module.
PEEK.POKE.....PEEKing and POKEing two-byte values.
COPY.FILE .....Copy any file from one directory to another.
ERROR.HANDLER.....A better way to fix Applesoft's ONERR bugs.
SET.DATE .....Sets ProDOS's date on systems without a clock.

I/O COMMANDS: PR# / IN#

MicroDot does not hook itself up to the I/O hooks except when a read or write mode (&.R or &.W) is active. Thus, the standard, pre-disk drive commands IN# and PR# work fine with *MicroDot*.

Do NOT precede these commands with an ampersand. Instead, use the commands in the same fashion as follows:

```
10 PR#3 : PRINT CHR$(25); : REM Activate 80 column card
20 IN#2 : REM Get input from modem
```

You can also redirect input and output by POKEing the KSW/CSW vectors in page zero. Location 54 and 55 ($36-$37) hold the address of the curent output routine; locations 56 and 57 ($38-$39) hold the address of the current input routine. As usual, the addresses are stored in low-high order.

Do NOT use PR# or IN# (or the above POKEs) while &.R or &.W is in effect. Instead issue &.Z first, then use IN# or PR#. Resume the read or write mode with &.R or &.W.

Use POKE 54,07 : POKE 55,195 to re-activate 80 column output after printing on a printer or other device. This is the equivalent to the BASIC.SYSTEM PR#A$3C07 command.

You might notice that screen output is a little faster under *MicroDot* than is was under BASIC.SYSTEM as *MicroDot* does not intercept each character as it is output.

THREE BONUS ROUTINES!

FLOPPY DISK FORMATTER

Introduction

It is a real drag that Apple charges a yearly fee to license their formatter. It is still another drag that even public domain (and other) formatters require you to quit your application, format a disk, then reboot your application. Not any more! We have made some changes to Jerry Hewett's public domain program HyperFORMAT from Living Legends Software. To be specific, we have relocated it to a "handier" location; your choice of three different locations in fact. And we have changed it so you can call the routine from within any BASIC program and it will return to your BASIC program just like you would want it to.

The formatter will format 5.25"disks and 3.5" disks. It will NOT format RAM disks or hard disks. We "locked out" these devices as a safety measure. If you really want to format these devices, use the finder or the system utilities. A disk formatted by the formatter can be made "bootable" by copying PRODOS and MICRODOT.SYSTEM (or any SYS file) to the new disk.

Using the Formatter

There are three versions of the FORMAT utility on the *MicroDot* disk. They differ only in their loading address/entry point. The versions are:

Filename	Memory used
FORMAT.1	$2000 (8192) - $3FFF (16383)
FORMAT.2	$4000 (6384) - $5FFF (24575)
FORMAT.3	$6000 (24576) - $7FFF (32767)

To format a disk, first load the disk formatting program into memory with the &.BL command. Then call the disk formatter with the following command:

```
CALL 8192,S,D,N$    (for FORMAT.1)
CALL 16384,S,D,N$   (for FORMAT.2)
CALL 24576,S,D,N$   (for FORMAT.3)
```

where S is the slot of the drive to use, D is the drive number, and N$ is the name desired for the new volume. The name should not include the slashes (e.g., "BLANK" instead of "/BLANK/"). FORMAT checks to be sure that you have specified a valid slot and drive but does not check to make sure that the volume name is legal. FORMAT also does not check to make sure the disk is blank or unformatted; you will can to do that from BASIC using &.V .

Both of the FORMAT routines use exactly 8K of memory. This means that they fit snugly inside the hi-res pages. The programs on the disk are quite a bit shorter than 8K, but they need a large additional workspace.

IMPORTANT ... IMPORTANT ... IMPORTANT ... IMPORTANT

NEVER reset the computer while disk formatting is going on. It will trash the disk, of course, but it will also mess up the computer's memory. The programs need to move quite a few things around in memory and they are careful to put everything back where it belongs when the formatting is complete, but if you press reset while the disk is spinning they will not have a chance to restore everything to normal and your computer's memory will be addled (which means worse than scrambled), particularly if your program uses memory in the $6800-$7FFF range.

If an error occurs during formatting, an error code $14 (20) will be returned via the normal error-handling mechanism. See the BASIC program FORMATTER on the disk for an example of how to use FORMAT from a BASIC program, and also as an example of a driver.

LINE EDITOR

Most commercially available program editors (such as GPLE, Edit Pro, etc.) were designed for use with BASIC.SYSTEM and do not work with *MicroDot*. Program Writer DOES work with *MicroDot*. See page 47 for a patch. We are currently investigating ways to modify other editors so that they can be used with *MicroDot*, but in the meantime, we have included a special version of a public-domain line editor called LIGHT (written by Jerry Kindall), to help you out a little. This program is on the disk as EDIT.

EDIT is not a full-featured program editor. It does not have global find and replace, macros, or renumber/merge. It does, however, make the task of making minor changes to program lines a lot easier.

To install EDIT, use the command &.BX,"EDIT". EDIT will load and connect itself to the ampersand hook. EDIT's use of the ampersand does not in any way interfere with *MicroDot*'s use of it. Once EDIT is installed, you can activate it by simply typing an ampersand. EDIT will connect itself to the Apple's KSW and CSW vectors in page zero and its edit keys will become "live" at all times.

EDIT's edit keys are:

Control-I (Tab): Insert a blank space at the cursor.
Control-D: Delete character under the cursor.
Control-O: Enter normally-illegal control character.
Delete: Delete character to the left of the cursor.

To edit an existing line, use the LIST command to list it on the screen. Then use the standard escape keys to move the cursor to the first digit of the line number. Now press <ESC.>. Next, use the right-arrow key to move to the first error and use the edit keys to correct it. You can use the left-arrow and right-arrow keys to go to any remaining errors and correct them with the edit keys. When you have made all your changes, use the right arrow key to trace to the end of the line and press the return key.

&XX: What about all those excess spaces Applesoft always inserts in program listings? Well, you could do a manual POKE 33,33 (and

that works best for editing REM and DATA statements) or you could use EDIT's compressed listing mode. To use that, simply type & followed by a line number, like this: **&10**. EDIT will do an automatic POKE 33,33 and list out the line with all non-quoted spaces compressed from the listing. This makes editing long lines much easier. Try it and see how it looks.

EDIT displays all control characters in listings (except control-M [return], control-H [backspace], and control-G [bell]) as inverse letters, so that you can trace over them in a listing and they will still be there. You can also use control-O to insert a control character into any line. The cursor will stop blinking and the next control character you type will be entered directly into the input buffer.

EDIT resides in page 3 of RAM, which means that it doesn't take any memory away from your BASIC workspace. Unfortunately, page 3 is a popular place for utility programs. EDIT can also cause problems if it remains connected through a program's GET or INPUT statements. For this reason, you should always disconnect EDIT before running a program.

The easiest way to disconnect EDIT is to press control-reset. PR# 0 : IN# 0 : TEXT will also do the trick. Once you're back in immediate mode, you can re-connect EDIT simply by typing an ampersand, unless your program has used page 3, in which case you should re-execute EDIT from disk. If you don't want to use the ampersand command, or if you are using the ampersand for something else, you can use EDIT via the CALL command. CALL 771 is the same as typing & by itself, and CALL 768,number is the same as typing &XX. If you want to use EDIT in this manner, you don't need to install it with &.BX; a simple &.BL will do fine.

EDIT also supports 80-column mode on the IIc, IIgs, and enhanced IIe. (Non-enhanced IIe's are not supported.) To use EDIT in 80-column mode, first enter PR#3. This will disconnect EDIT, so enter & to re-connect it. You can then use PR#3 instead of control-reset or PR# 0: IN# 0: TEXT to disconnect EDIT. In 80-column mode, EDIT does not show control characters in inverse. It does, however, set a 72-column wide window when doing a compressed listing.

PROGRAM WRITER PATCH

The Software Touch's *Program Writer*, by Alan Bird, is, in our opinion, absolutely the best BASIC program editor available. In case you haven't seen it, it's a full-screen editor as opposed to a line editor like GPLE or the EDIT program included on this disk. You can scroll backwards and forward through the program, changing lines simply by putting your cursor where you want to make a change and typing the correction. It's like AppleWorks' Word Processor, for BASIC programs. It also has built-in auto line numbering, renumbering, variable cross reference, macros, and mouse support. If you don't have it already we recommend it highly.

If you do already have it, though, you're probably wondering if it can be used with *MicroDot*. Well, the standard version of Program Writer doesn't work with *MicroDot*, but there's a utility on the *MicroDot* disk called PW.ADAPTER that patches Program Writer to work with *MicroDot*.

PATCHING PROGRAM WRITER

1) Boot the *MicroDot* system disk. When the Applesoft prompt appears, enter &.X,"PW.ADAPTER" and press the return key.

2) Put a copy of your original Program Writer disk in any drive and enter the number of the editor to patch, as displayed on the screen. These editors are EDITOR, the main-memory version of Program Writer; EDITOR.LC, the aux-mem language-card version (requires 128K), and EDITOR.SMALL, the stripped-down main memory version. NOTE: In order for PW.ADAPTER to work, the name of your Program Writer disk must be /EDITOR/, which is how it is supplied by The Software Touch (which is now part of Beagle Bros.).

3) After you have chosen an editor to patch, the appropriate file will be loaded into memory and patched. PW.ADAPTER is written in BASIC and is designed to work with ALL versions of Program Writer, even those that haven't been released yet, so it will take a

few minutes to search through the editor and change everything that needs changing.

4) When patching is completed, the patched editor will be saved onto the *MicroDot* disk as PW, PW.LC, or PW.SMALL, depending on which editor you were patching. These PW files work only with *MicroDot* and not with BASIC.SYSTEM.

5) If you only have one disk drive, you will naturally have to remove the *MicroDot* disk, insert the Program Writer disk to load the editor, then re-insert the *MicroDot* disk to save the patched version of Program Writer. You aren't prompted to do this. Simply remove the *MicroDot* disk and insert the Program Writer disk before selecting an editor, then, after the editor has been loaded, put the *MicroDot* disk back in the drive. You have plenty of time to do this while patching is in progress.

USING PROGRAM WRITER

To install Program Writer, simply enter &.BX,"PW" (or use "PW.LC" or "PW.SMALL" as appropriate). Then, to invoke Program Writer, just type && at the Applesoft prompt as usual. To leave Program Writer and return to immediate mode, enter OA-Q as usual.

The OA-G (Get Macros) and OA-S (Save Macros) commands are disabled in the *MicroDot* version of Program Writer. This is because Program Writer calls BASIC.SYSTEM routines which don't exist in *MicroDot*. You should not use the OA-Z command to remove Program Writer from memory, either; instead, re-execute MICRODOT.SYSTEM.

The configuration program supplied with Program Writer will not work with the patched versions of Program Writer. You should configure only standard, Software Touch supplied versions of Program Writer, **then** patch them with PW.ADAPTER. (The configuration program is used to install a default macro set into Program Writer, among other things.) Be sure to save the configured editor back onto the Program Writer disk using the standard editor name.

TECHINICAL INFORMATION

AMPERSAND COMPATIBILITY

Since *MicroDot* uses the ampersand to intercept its commands, it might seem that you can't use ampersand routines with *MicroDot*. But *MicroDot* does provide a way to hook in other ampersand routines. At locations $3E8-$3E9 (1000-1001) is *MicroDot*'s new ampersand vector. If you are writing a new ampersand routine to work with *MicroDot*, you can hook it into these two bytes. Unlike Applesoft's usual ampersand vector, which is a three-byte jump statement (a $4C followed by a two byte address), *MicroDot*'s ampersand vector is simply a two byte address.

It should also be possible to modify existing ampersand routines to work with *MicroDot*, as long as they do not call any BASIC.SYSTEM routines. Simply look through the code searching for instructions that address locations $3F5, $3F6, and $3F7 (the normal ampersand vector). Remove (by replacing with NOP's, $EA) any references to $3F5, and change references to $3F6 and $3F7 to $3E8 and $3E9, respectively.

Many ampersand routines call BASIC.SYSTEM to get an area of memory for the program, or for some other purpose. These will not work with *MicroDot* without extensive modification.

MICRODOT STARTUP PROCEDURE

Here's a step-by-step description of what *MicroDot* goes through when it is executed.

First, if the computer has an 80-column card, *MicroDot* deactivates it by sending a control-U character to the card.

Next, Applesoft BASIC is started up by entering it at $E000. *MicroDot* regains control by attaching itself to the I/O hooks, CSW and KSW.

Then the video display is set to default conditions. Keyboard input, screen output, text mode, full-screen display, and normal (as opposed to inverse) display mode are activated. The screen is cleared.

The *MicroDot* program itself is then moved to $B300-$BAFF, and the ampersand vector and reset vector are set to point to *MicroDot*. HIMEM is lowered to protect *MicroDot* and the page 3 locations are initialized. The stack pointer is set to $FF.

The ProDOS memory map is altered to protect pages 0, 1, 4-7, $B3-$BA, and $BF. All other memory pages are left unprotected.

The title screen is printed.

If a null prefix exists, the prefix is set to the volume name of the current disk.

Finally, the startup program is executed. Supported filetypes include BAS, BIN, SYS, and TXT (exec files). If no startup program exists, *MicroDot* simply exits to Applesoft BASIC.

MICRODOT RESET HANDLING

When you press control-reset, *MicroDot* gets control of the computer. Usually it simply passes control back to BASIC, but you can change this by storing a new address in *MicroDot*'s reset vector at $3EA-$3EB (1002-1003).

To make reset pass control to an ONERR GOTO routine in your program, simply store the address $E30B in the reset vector. (POKE 1002,11: POKE 1003,227.) This will cause *MicroDot* to issue an Applesoft ?ILLEGAL DIRECT ERROR (code 149) when reset is pressed. This error can never occur naturally within a running program, so if you get an error code 149 in your ONERR handling

routine, you know reset has been pressed. You cannot simply RESUME to your program when reset is pressed; too many things in Applesoft get changed to allow this. Instead you should use a GOTO statement to get back to your main program loop. Also remember that pressing reset turns off graphics mode, the 80-column card, and any other peripherals that are turned on, so you may need to take some special steps to re-initialize the display.

MicroDot usually jumps to location $D43F when RESET is pressed. To turn off reset trapping, just store this address back into the *MicroDot* reset vector. (POKE 1002,63: POKE 1003,212.) If you would like RESET to re-RUN the program from the first program line use POKE 1002,102 : POKE 1003, 213.

MICRODOT ERROR HANDLING

MicroDot, for the most part, uses standard and Applesoft ProDOS MLI error codes. A few nonstandard error codes were implemented for use by *MicroDot* itself to indicate errors which are not supported by ProDOS. All errors can be trapped by ONERR GOTO.

COMMAND PARSING ERRORS

If an error is encountered while parsing (i.e., figuring out what is meant by) a command, *MicroDot* will issue a normal Applesoft error like ?SYNTAX ERROR, ?ILLEGAL QUANTITY ERROR, ?TYPE MISMATCH ERROR, and ?STRING TOO LONG ERROR. These error messages indicate that something is wrong with the command itself. The standard error numbers will be placed in location 222, the usual error-number location.

COMMAND EXECUTION ERRORS

The rest of *MicroDot*'s errors occur when it tries to complete a command but is prevented from doing so by a ProDOS error. For example, if you tried to load a program and there was a bad block in the disk file, you would get one of these errors.

Standard MLI error codes are passed back via *MicroDot*'s error handling routine. These error codes are much more specific than BASIC.SYSTEM's. This is because BASIC.SYSTEM's goal is to be easy to understand. On the other hand, *MicroDot* gives you much more flexibility in handling errors because you know exactly what happened.

When an error occurs, the number one (1) is placed in location 222. If your ONERR handling routine encounters this value in PEEK(222), it knows that a *MicroDot* error occurred. The actual error number can then be obtained by PEEKing location 993. Here is a list of the errors you can encounter:

Hex	Dec	Error
$01	001	Invalid MLI function code number. You will get this if you pass an invalid MLI function code to the SYSCALL module.
$04	004	Incorrect parameter count in parameter list. This may be a result of a bad parameter list in a SYSCALL (&.Y) command.
$10	016	File type mismatch. The file is of the wrong type.
$11	017	Program too large to fit into memory.
$12	018	No buffer available for this file. (MULTI module.)
$13	019	Files still open; cannot change buffer allocation. (MULTI module.)
$14	020	Unable to format disk properly. (FORMAT utility.)
$25	037	The interrupt table is full. You will get this if you attempt to install more than four interrupt handlers. You cannot do this from *MicroDot*, but SYSCALL will do it.
$27	039	I/O error. Could be anything from an open drive door to a bad disk, or, with floppy drives, no disk in the drive.
$28	040	No device connected for the unit number given.
$2B	043	The disk is write-protected; unable to write to the disk.
$2E	046	A diskette for which there were open files has been removed from a drive. Some blocks on the new volume may be damaged.
$40	064	Invalid pathname syntax. May also indicate a null prefix.

47 No Disk in drive (3½" only)

Hex	Dec	Error
$42	066	Eight files are already open and no more can be opened. You will never get this ProDOS error; the MULTI module will return an out-of-buffers ($12/18) error instead.
$43	067	The reference number specified does not denote an open file.
$44	068	The pathname specified could not be followed. Subdirectory not found.
$45	069	The volume specified could not be found.
$46	070	The file specified could not be found.
$47	071	The file already exists. (&.M or &.N error.)
$48	072	The disk is full. No more information can be stored on it.
$49	073	The volume directory is full. It can hold a maximum of 51 files. Note this is different from disk full.
$4A	074	Version of ProDOS too old to access this file.
$4B	075	Bad storage type. The directory is probably damaged.
$4C	076	Unable to read past the end of the file.
$4D	077	Unable to move the file pointer past the end of the file.
$4E	078	File is locked, or error in access bits.
$50	080	The file specified is already open. Multiple OPENs on the same file are allowed only if the file is locked.
$51	081	File count in error; directory is probably damaged.
$52	082	Not a ProDOS disk.
$53	083	One of the parameters for an MLI call is out of range. (SYSCALL module.)
$55	085	More than eight volumes cannot be mounted at once.
$56	086	Bad disk buffer address.
$57	087	Two volumes with the same name are online.
$5A	090	The volume bit map of the disk is damaged.

When using an AppleShare network, additional error codes may be valid. Unfortunately, information about these additional errors are not available at the time of this writing. Contact Apple II Developer Technical Support.

MICRODOT PEEKS AND POKES

This is a list of PEEKs and POKEs which you may find useful in conjunction with *MicroDot*.

GENERAL

PEEK (46018) + PEEK (46019) * 256.....**Address of last BLOAD**
PEEK (46022) + PEEK (46023) * 256.....**Length of last BLOAD**
PEEK (996) + PEEK (997) * 256.....**CSW Vector** (Save area during &.R)
PEEK (998) + PEEK (999) * 256.....**KSW Vector** (Save area during &.W)
PEEK (45885)...**Reference number of last file opened**
PEEK (1004).....**Last character read by &.R (See &.R)**
PEEK (992).....***MicroDot* version number (0 is first version)**
PEEK (993).....***MicroDot* error code number**

POKE 46047,0 : POKE 49044,0 : & .C.....**Perform global close** (p. 9)
POKE 1000,LOW : POKE1001,HIGH .....**Ampersand (&) Vector** (p. 49)
POKE 1002,LOW : POKE1003,HIGH .....**Reset Vector** (p.50)
POKE 54, LOW : POKE 55, HIGH.....**Output Vector** (p.42)
POKE 56, LOW : POKE 57, HIGH.....**Input Vector** (p.42)
POKE 54,PEEK(996) : POKE 55,PEEK(997)...**Echo READs to screen** (p.35)

FILE ATTRIBUTES

Valid immediately after &.IG,NAME$,TYPE,AUX
Can also be POKEd immediately before &.IS, NAME$,TYPE,AUX

PEEK (46001)..**Access Byte** - In general 195 ($C3)= Unlock, 1=Locked
.....Bit 7 = 1 **Allows file to be deleted**
.....Bit 6 = 1 **Allows file to be renamed**
.....Bit 5 = 1 **File needs to be backed up**
.....Bit 1 = 1 **Allows file to be written to**
.....Bit 0 = 1 **Allows file to be read**
.....Bits 4, 3, and 2 are **Unused**
PEEK (46002) .....**Filetype** (Same as returned by &.IG)

FILE ATTRIBUTES (cont'd)

PEEK (46005) .....**Storage**
.....1 = seedling file (no index blocks)
.....2 = sapling file (one index level)
.....3 = tree file (two index levels)
.....13 = directory file (linked)
PEEK (46003) + PEEK (46004) * 256 .....**Auxtype** (same as returned by &.IG)
PEEK (46003) + PEEK (46004) * 256 .....**# of Blocks on Volume**
If &.IG a Volume Directory
PEEK (46006) + PEEK (46007) * 256 .....**# of Blocks used by file**
PEEK (46006) + PEEK (46007) * 256 .....**# of Blocks used on Volume**
If &.IG a Volume Directory
PEEK (46008) + PEEK (46009) * 256.....**Modification Date**
(format: YYYYYYYYMMMMDDDDD)
PEEK (46010) + PEEK (46011) * 256.....**Modification Time**
(format: HHHHHHHHMMMMMMMM)
PEEK (46012) + PEEK (46013) * 256.....**Create Date** (Can't be POKEd)
(format: YYYYYYYYMMMMDDDDD)
PEEK (46014) + PEEK (46015) * 256.....**Create Time** (Can't be POKEd)
(format: HHHHHHHHMMMMMMMM)

FILE ATTRIBUTES (Directory Entries)

Valid immediately after &.G,NAME$,TYPE,AUX

PEEK (640).....**Storage Type / Name Length**
(Format: SSSSNNNN)
PEEK (641-655).....**Filename** (Same as returned by &.G)
PEEK (655).....**Type** (Same as returned by &.G)
PEEK (656) + PEEK (657) * 256.....**Pointer to key block**
PEEK (658) + PEEK (659) * 256.....**# of Blocks used by file**
PEEK (660) + PEEK (661) * 256 + PEEK (662) * 65536.....
Length of file in bytes (EOF)
PEEK (663) + PEEK (664) * 256.....**Create Date**
(format: YYYYYYYYMMMMDDDDD)
PEEK (665) + PEEK (666) * 256.....**Create Time**
(format: HHHHHHHHMMMMMMMM)

FILE ATTRIBUTES (Directory Files) cont'd

PEEK (672) + PEEK (673) * 256.....**Modification Date**
(format: YYYYYYYMMMMDDDDD)
PEEK (674) + PEEK (675) * 256.....**Modification Time**
(format: HHHHHHHHMMMMMMMM)
PEEK (667) .....**Version of ProDOS which created file**
PEEK (668).....**Minimum version of ProDOS which can access file**
PEEK (669).....**See access bits definition above** [PEEK (46001)]
PEEK (670) + PEEK (671) * 256.....**Auxtype** (Same as returned by &.G)
PEEK (676) + PEEK (677) * 256.....**Header** (Pointer to file's parent directory)

MICRODOT GLOBAL LOCATIONS

This section details the *MicroDot* global locations. These addresses are most useful from machine language subroutines, but some may be useful from BASIC; they are a complete set of tools for development of ML routines that work with *MicroDot*.

MicroDot Entry Points

AMPENT	$B300	Standard entry to *MicroDot* when & is entered.
ALTENT	$B303	CALL entry to *MicroDot*. CALL instead of &. CALL 45827,cmd,parms
MAKE	$B306	Make a file; pathname at $280; type at $1A
FIND	$B309	Find existing file; path at $280; type at $1A
TEST	$B30C	Test for/make file; path at $280; type at $1A
READ1	$B30F	Read one byte into accumulator & loc $3EC
READ	$B312	Read many bytes; uses PRW parmlist
WRITE1	$B315	Write one byte from accumulator
WRITE	$B318	Write many bytes; uses PRW parmlist
GETMARK	$B31B	Get position in file; returned to PPOS parmlist
SETMARK	$B31E	Set position in file; from PPOS parmlist
GETEOF	$B321	Get end-of-file; returned to PPOS parmlist
SETEOF	$B324	Set end-of-file; from PPOS parmlist
OPEN	$B327	Open file; uses POPEN parmlist and path at $280

CLOSE	$B32A	Close file; uses PCLOSE
IOREAD	$B32D	Hook read up to input; same as &.R
IOWRITE	$B330	Hook write up to output; same as &.W
IOSTOP	$B333	Turn off &.R or &.W mode
SETDIR	$B336	Set up to begin reading directory
GETDIR	$B339	Read next directory entry to $280
GETINFO	$B33C	Get file attributes; uses PINFO
SETINFO	$B33F	Set file attributes; uses PINFO
DELETE	$B342	Delete file; pathname at $280
RENAME	$B345	Rename file; current path at $2C0, new at $280
KILL	$B348	Kill to end of file (same as &.K)
APPEND	$B34B	Move position-in-file to eof
GETPFX	$B34E	Get current prefix to $280
SETPFX	$B351	Set prefix to path at $280
VOLUME	$B354	Get name of volume in unit in PONLINE parmlist
QUIT	$B357	Normal ProDOS bye
GOSYS	$B35A	Execute system file; path at $280
CHKERR	$B35D	Check carry for error after MLI call
DOERR	$B360	Force *MicroDot* error processing
GETNUM	$B363	Parse a 2-byte number into A/X & $1A/$1B
SETNUM	$B366	Pass a 2-byte number back to variable
GETSTR	$B369	Get a string (path) to $280
SETSTR	$B36C	Pass string back from $280 to variable
RELOC	$B36F	Adjusts link bytes of the program starting at the adress pointed to by $1A, $1B
FUTURE	$B372	For future expansion

MLI Parmlists

PPOS	$B3A9	Used by GETMARK, SETMARK, GETEOF, SETEOF
PINFO	$B3AE	Used by GETINFO, SETINFO
PRW	$B3C0	Used by READ, WRITE
PONLINE	$B3C8	Used by VOLUME
PCREATE	$B3CC	Used by MAKE
POPEN	$B3D8	Used by OPEN
PCLOSE	$B3DE	Used by CLOSE
PDEST	$B3E0	Used by DELETE

PRENAME	$B3E3	Used by RENAME
PPFX	$B3E8	Used by GETPFPX, SETPFX
PRW1	$B3EB	Used by READ1, WRITE1

Variables

CMDLO	$B375	Table of lo-bytes of command handlers
CMDHI	$B38F	Table of hi-bytes of command handlers
VERSION	$3E0	*MicroDot* version, first version is zero
ERRCODE	$3E1	*MicroDot* error code
HIGHEST	$3E2	Highest legal HIMEM (Used by modules during installation)
XCSW	$3E4	Previous CSW before &.R/&.W
XKSW	$3E6	Previous KSW before &.R/&.W
NEWAMP	$3E8	New ampersand vector for *MicroDot*
NEWRES	$3EA	New reset handling vector for *MicroDot*
CURRENT	$3EC	Current directory entry, or current character
EXTFLG	$3ED	User error flag; If > 128 errors will go to EXTERR
EXTERR	$3EE	Address of user error handler

If an error occurs when executing one of the $B3xx routines, it will be passed back to BASIC as usual **unless** you set the EXTFLG and EXTERR locations properly. If you hook up your own error trapping, be sure to save the stack pointer on entry to your routine and restore it before RTSing to BASIC. Also, make **sure** you turn off EXTFLG on exit. When your error handler receives control, the MLI error code will be in the Accumulator.

Turn on External Error Handling

```
LDA ....# HIBYTE ; of user handler
STA ....EXTERR + 1
LDA ....# LOBYTE ; of user error handler
STA ....EXTERR
SEC
ROR.....EXTFLG
```

Turning off External Error Handling

```
LSR.....EXTFLG
```


PRODOS FILE TYPES

This is a semi-complete listing of file types defined under ProDOS. Types not listed are reserved for future use, or were used in the past by SOS, the now-extinct Apple /// DOS. Types denoted "IIgs" or "ProDOS 16" are used only by ProDOS 16 or GS/OS the IIgs operating system. Notations in parentheses may denote alternate uses for a given filetype.

Type	Hex	Dec	Description
NUL	$00	0	Null file (unused)
BAD	$01	1	Bad Block or file
TXT	$04	4	Standard Text File
BIN	$06	6	Standard Binary File
DIR	$0F	15	Subdirectory File
ADB	$19	25	AppleWorks Data Base
AWP	$1A	26	AppleWorks Word Processor
ASP	$1B	27	AppleWorks Spreadsheet
SRC	$B0	176	APW Source File (IIgs)
OBJ	$B1	177	APW Object File (IIgs)
LIB	$B2	178	APW Library (IIgs)
S16	$B3	179	ProDOS 16 System Program (IIgs)
RTL	$B4	180	APW Runtime Library (IIgs)
EXE	$B5	181	APW Shell program (IIgs)
PIF	$B6	182	ProDOS 16 permanent init file
TIF	$B7	183	ProDOS 16 temporary init file
NDA	$B8	184	New Desk Accessory (IIgs)
CDA	$B9	185	Classic Desk Accessory (IIgs)
TOL	$BA	186	ProDOS 16 Tool Set
DRV	$BB	187	ProDOS 16 Device Driver
DOC	$BF	191	ProDOS 16 Document
PNT	$C0	192	Packed SHR Picture (IIgs)
PIC	$C1	193	Unpacked SHR picture (IIgs)
FON	$C8	200	ProDOS 16 Font
FND	$C9	201	ProDOS 16 FINDER data
ICN	$CA	202	ProDOS 16 Icon
DDD	$DD	221	Reserved (DDD Packed File)
BLU	$E0	224	Reserved (Floyd Zink's new BLU?)
DTS	$E2	226	ATINIT - AppleTalk Init File?
PAS	$EF	239	Pascal partition on disk
CMD	$F0	240	ProDOS 8 command file
USR	$F1	241	ProDOS 8 User Defined (ProBASIC Program)
USR	$F2	242	ProDOS 8 User Defined (ProBASIC Module)

USR	$F3	243	ProDOS 8 User Defined
USR	$F4	244	ProDOS 8 User Defined
USR	$F5	245	ProDOS 8 User Defined
USR	$F6	246	ProDOS 8 User Defined
USR	$F7	247	ProDOS 8 User Defined
USR	$F8	248	ProDOS 8 User Defined (Merlin LNK)
P16	$F9	249	ProDOS 16 Operating System
INT	$FA	250	Int. BASIC Program (Beagle Compiler COM)
IVR	$FB	251	Int. BASIC Variables (Beagle Compiler CVR)
BAS	$FC	252	AppleSoft BASIC Program
VAR	$FD	253	AppleSoft BASIC Variable
REL	$FE	254	EDASM Relocatable Code Module
SYS	$FF	255	ProDOS 8 System Program

HELP !

Got a technical question? Give us a call at (614) 891-2111. If we can't answer your question, we'll give you the author's phone number. He will be able to answer your questions. If you use GEnie you can write directly to the author (address: J.KINDALL).

INDEX