DotBASIC Plus

the ultimate hobbyist programming language for the commodore 64

Another Quality Product from

LOADSTAR

DotBASIC PLUS

A Compleat Guide to the Ultimate Programming Environment for the Commodore 64

DotBASIC Plus created by
Dave Moorman

Based on Mr.Mouse Technology
by Lee Novak

Manual by
Dave Moorman, Lee Novak, Alan Reed

Join the DotBASIC Community

http://8bitcentral.com/dotbasic

More from these Authors:

"Shall Not Perish" or "The god of 524288 Switches", Salvific Issues of Automatic Data Processing
by Dave M. Moorman

Also Steal This Book
by Alan Reed

Untitled: A Tale of Indecision
by Lee Novak

Copyright 2008 eTower Marketing
All Rights Reserved
All Programs Copyright 1982-2008 J&F Publishing, Inc

USING THIS BOOK

We at **LOADSTAR** are pleased to offer the ultimate software development package for the Commodore 64: **DotBASIC Plus**.

DotBASIC Plus (**DB+**) is a new and exciting way to put pizzazz in your programs. While **DotBASIC** is astoundingly easy to use, there are a few new concepts that may be unfamiliar to the **BASIC 2.0** programmer. So, we highly recommend you start from the beginning. The first part of the manual will walk you through a number of program ideas, explaining the various **DB+** commands and concepts along the way. Take the time to work your way through them—we promise it will pay off! Once you get the hang of **DotBASIC**, you'll be amazed at how slick and professional your programs can look with just a few lines of code.

> "DotBASIC is always growing! Visit the DotBASIC FORUM often for new DotCommands and other updates!"
>
> www.8bitcentral.com/dotbasic/forum
>
> LOADSTAR TIP ★

DotBASIC is the ideal programming environment for the novice – however, at least some familiarity with **BASIC 2.0** (the built-in language on the C-64) will help. We will start from scratch. Yep — *"Hello, World"* will be our first project. From there, we will explore doing text boxes, menus, Event Regions, mouse controls, and a host of other features.

The second part, ***The DotBIBLE***, is the "Reference Guide" section of the manual. Every current **DotBASIC** command (cleverly abbreviated to *DotCommand*) is listed here alphabetically. You'll be able to look up the proper syntax and see a short description of each DotCommand.

In the ***Appendices*** you'll find a **DotBASIC** Quick Reference sheet, a DotCommand Summary, info for machine language programmers, and other interesting and useful things for the more advanced user.

The following conventions are used throughout this book:

- Special Commodore characters are shown in brackets. For example, **{clr}** is the SHIFT / HOME key combination, **{down}** is the **CRSR** down key, and so on.
- We will use the "*variable*" ***db*** for the disk drive containing your **DotBASIC** disk, as in **LOAD"B.DEV",*db***. Likewise, ***dw*** represents the drive containing your Work disk.
- We present DotCommands in the following way:

  NAME

  Syntax: ***dotcommand,parameters***

  Some DotCommands are easier to remember than others. **.MENU** is pretty self explanatory, but "Random Index" makes a little more sense than **.RDMI**—thus the blue *NAME* field.

  Some typical parameters used throughout:

  X,Y,W,H This is the *area* field. **X** and **Y** represent row (0-39) and column (0-24) of the upper left corner of the area. **W** is *width* (1-40) and **H** is height (1-25).

  CO Color

SC Screen Code

D Disk drive device number. Your current drive number is automatically assigned to **D** by your **DotBASIC Plus** program. There's no longer any need to worry about what device number to use when creating programs that require disk drive access. Just use **D.**

LOC Location in RAM memory, as in 49152 or 40960.

PAGE Page number in RAM. Each page is 256 bytes long. To convert a page to a decimal memory location, multiply by 256. For example, page 192 is 192*256 = 49152. We at **LOADSTAR** like using pages when talking about memory. Refer to page 6, *Making Memory Management Manageable*, to see why using Pages just makes sense.

STRING$** String, as in **A$, or **"STRING"**. Most DotCommands can use string variables (**A$**), literals (**"STRING"**) or concatenated strings (**A$+"STRING"**). A few, however, require a literal string and a few insist on string variable only.

SYSTEM REQUIREMENTS

Keep your original **DB+** disks in a safe place. Use a copy of the **DotBASIC Plus** disk and at least one disk drive. The system really works best if you have two or more drives. In fact, this manual assumes that you are using two drives. The **DB+** Library disk should be in one drive and your formatted *Work Disk* in the other. *(I use drive 8 for my Work and drive 9 for the **DB+** Library disk. Your situation may vary. I also work in **VICE, the Versatile Commodore Emulator**, on a Pentium 400 Windows 98 machine. Even on the shiny new equipment, I am woefully behind the cutting edge!—Dave)*

If you absolutely must use a single disk drive, you will need to be sure you have 100+ blocks free on your **DotBASIC Plus** disk to have room for your work. Just be sure to copy your program to another disk when the project is completed. Your essential files are: B.YOURPROG, MOUSE 2.1 7K 2000, YOURPROG.DML, YOURPROG.DBS, and any file with a DBA prefix. Did we mention that **DotBASIC** really works best with two disk drives?

DOTBASIC FEATURES

1. **DotBASIC Plus** is an Adaptable, Mouse Capable BASIC Extension. **DB+** programs boot from default BASIC, and exit to default BASIC. **DB+** currently has 100 additional commands, using an easy-to-use and easy-to-read ***".command"*** structure. Your **DB+** program has code overhead for only the DotCommands you use.
2. Quick environment set-up puts all the necessary files on your work disk and creates a boot program for your project. Easily "include" the DotCommands you need with "DEV". Library commands are merged with your program code.
3. Programs written with **DotBASIC Plus** automatically support a mouse in port 1 and a joystick in port 2. The joystick FIRE button is the same as the left mouse button. For joystick users, any key can be defined to replace the missing right mouse button. If you have CMD's SmartMouse, the middle button will double the mouse's speed.
4. *Do-Loop* Method for Fast Mouse Control, with *Loop-Until* and *Loop-While*. Event Regions and Mouse Roll-Over effects, including "Region Text" to display information.
5. Boxes/Windows, Screen Stash/Restore, Text Area Cut/Paste for pop-in/pop-out menus, dialogs and effects. Print At/Print Center/Word-Wrapped Text Blocks. Wait for Key/Wait for Key or Mouse Click. Read Keypress against given string of keystrokes. Two improved INPUT commands.
6. BLOAD/BSAVE/Directory Load/Disk Command. Character/Color Swap/Copy Memory/Swap Memory
7. Regular Menus/Scrolling Menus/Multi-Column Menus/Scrolling-Multi-Select Menus.
8. Rack ***Mr.Edstar*** (**LOADSTAR's** 38-column editor) files into virtual string arrays — even under ROM. Store strings in memory (under ROM) and Rack into a virtual array.
9. **SIDPlayer!** Craig Chamberlain's classic ***Enhanced SIDPlayer*** available as two DotCommands. Load and display High-Res and Multi-Color Bitmap Graphics. Draw on Bitmap Screen. Print Text on Bitmap Screen. Sprite Control, including expansion, priority, and multi-color mode effects. Link sprites' positions to each other for "easy-to-place" mega-sprites.
10. ***AND THE GREATEST FEATURE OF ALL***— **DB+** is designed to grow, with new commands and variations being continuously developed. Just add the new command files to your library and use them as needed.

GETTING STARTED

IN THIS SECTION: **Page**

Creating a New Project	7
DotCommands	
TEXT COLOR	8
BACKGROUND COLOR	8
BORDER COLOR	8
OFF	8
DO-LOOP	9
MOUSE ASK	9
UNTIL	9
"Including" New DotCommands	10
Creating Text Boxes	12
DotCommands	
TEXT BOX	12
BOX	12
KEY/MOUSE WAIT	12
Regions	13
DotCommands	
DEFINE REGION	13
AFFECT REGION	14
PUT MOUSE	14
PRINT AT	14
PRINTCENTER	14
KEYPRESS	16

MAKING MEMORY MANAGEMENT MEMORABLE

At **LOADSTAR** we like simplicity. But memory locations are quite confusing, especially when written in decimal numbers. And, if you are rather new to programming, those hexadecimal numbers (like $A000) are even harder to read. So we use *Pages*. The 64K of the C-64 can be divided into 256 Pages, each containing 256 bytes. Actually, this is how the computer sees memory, described in two bytes — *lo byte* and *hi byte*. We call the hi byte the *Page number*.

Here are some Page numbers that are important with **DotBASIC Plus**:

0	Zero Page — where the system does a lot of work.
4	Beginning of the text screen.
8	Beginning of the font.
16	Beginning of DB+ code
46-54	Sprite image area.
55-??	DB+ commands.
160	Beginning of BASIC ROM
176	Half way through BASIC ROM
192	"Open" memory.
208-223	Input/Output registers
224-255	Kernal ROM

Pages 128-207 will be used with certain DotCommands, such as **.SID** and **.BMP**. Be sure to look at the Memory Map on the Quick Reference Sheet when using these commands. When we have a ***Mr.Edstar*** text to BLOAD, we just multiply the Page number by 256, as in:

```
.bl,"t.text",d,160*256
```

Another great trick: on the disk directory, we can see how many Disk Blocks a file uses. A Disk Block is 254 bytes long, so we can use this number to figure out how many pages a file will encompass. For instance, if "t.text" is 12 Blocks in size, we know we can BLOAD "t.text2" to page 160+12, or

```
.bl,"t.text2",d,172*256
```

Now that's memorable!

GETTING STARTED: CREATING A NEW PROJECT—“*HELLO, WORLD*”

That’s right, ladies and gentlemen, we are going to start with the semi-obligatory *“Hello, World”* program. This tutorial will introduce you to the world of **DotBASIC** programming, gently acquaint you with some programming concepts that might be new to you, and show you a few DotCommands. Perhaps most importantly, we’ll teach you how to create a new **DotBASIC** Work disk.

> “To watch a video version of the Hello World tutorial, visit the DOTBASIC FORUM”
>
> **www.8bitcentral.com/dotbasic/forum**
>
> LOADSTAR TIP ★

So are you ready to go? Format your Work disk, making sure all disks are in the proper drives. Remember, we will use the "*variables*" ***db*** for your **DB+** disk drive and ***dw*** for the Work disk drive. Ready?

Here we go!

LOAD"B.DOTBASIC",*db*

And

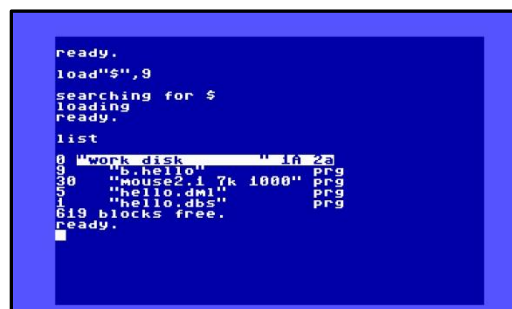
RUN

First, choose the drive your Library (“Dot”) disk is in. The next menu that appears is "Work Dsk." Choose the drive your Work Disk is in. You will be asked to confirm both choices— press **N**, then press **Y** to try again.

Creating a new DotBASIC project

You are now prompted to input the program name. We will call this program **hello**. Type in "hello"and press **RETURN**.

The necessary files for your project are created and copied to your Work drive. Then your project is automatically RUN. You will see the mouse arrow for a second, then a READY prompt in white. You are now ready to program in **DotBASIC Plus**! Let’s see what we have on our Work disk now. **LOAD"$",*dw*** And **LIST** the program. You will see:



B.HELLO is the boot program, the one you will load and run to start your DotBASIC program. The “B.” prefix is a **LOADSTAR** convention that makes it easy to recognize a multi-part program’s boot file. You can rename the boot file to anything you like, of course, but *do not rename any of the other files on your Work disk*.

MOUSE2.1 7K 1000 provides many of the commands you will use.

HELLO.DML is the ML program that interprets commands from **DB+** and makes them work. One of the neat things about **DotBASIC Plus** is that the .DML program only contains the DotCommands your program needs—you won't waste valuable RAM on DotCommands you aren't using *(see page 10 for more on this).*

HELLO.DBS is your **DB+** program. This is the place where you will be working.

Next, **LOAD"HELLO.DBS",*dw*** and **RUN** it. Now **LIST** the program:

```
5 d=peek(186): dd=56*256: mm=16*256
10 rem begin list
20 rem.endlist
30 sysdd
40 .tx,1: print"{clr}";:.bg,6:.br,14
59998 .of
59999 end
60000 gosub60008: n$=n$+".dbs"
60001 d=peek(186): sys14339
60002 open1,d,15,"s0:"+n$: close1
60003 saven$,d:end
60008 n$="hello"
60009 return
```

Line 5 sets up important variables. Lines 10 through 20 are used to add commands to your **DB+** program. Line 30 starts **DB+**. Line 40 uses three **DB+** commands.

When using a DotCommand immediately after THEN, use a colon first:

```
1000 if rg%= 5 then : .areg,5,255,7
```

LOADSTAR TIP ★

TEXT COLOR

Syntax: **.TX,CO**

.TX sets the Text color. Add 128 to **CO** (color) to REVERSE the text.

BACKGROUND COLOR

Syntax: **.BG,CO**

Sets the background color.

BORDER COLOR

Syntax: **.BR,CO**

Sets the Border color.

Sure, **POKE 53280,CO**, **POKE 53281,CO** or **POKE 646,CO** will work just fine. But aren't these commands easier to read? And the +128 option for **.TX** will come in handy. Promise!

On line 59998 we have another DotCommand:

OFF
Syntax: **.OF**

.OF turns off **DB+** and returns your computer to its normal default state. The **SYS14339** in line 60001 does the same thing.

When you break out of a **DB+** program and the arrow is still visible, **DotBASIC Plus** has NOT been shut off. You should type **.OF** in immediate mode to stop the special features. However, like me, you will forget. Therefore, before **RUN**ning your edited code, first **GOTO60000.** This is the beginning of our **LOADSTAR** "*scratch and save*" routine. With **DB+**, all you ever need to do to save your program is **GOTO60000** and press **RETURN**. This not only saves your current work, it also turns off **DB+**. If **DB+** is not turned off before **RUN**ning, the computer will usually lock up.

> **"Get used to typing GOTO60000 whenever you make changes."**
> LOADSTAR TIP ★

Impressed? Confused? Don't worry. We will now write *"Hello, World"*!

```
100 print"Hello, World"
102 .do
104 .ma
106 .un l2% or peek(198)
108 print"{clr}";
110 poke 198,0
```

Actually, the printing of "*Hello World*" is just plain **BASIC 2.0**. The **DB+** improvement comes with lines 102-106 and three new DotCommands:

DO-LOOP
Syntax: **.DO**

.DO begins a *Do-Loop*. If you have worked with any language other than **BASIC 2.0**, you know what a Do-Loop is all about. Much of your mouse control and effects can be performed within a Do-Loop, making your program very responsive. We will explain in a moment.

MOUSE ASK
Syntax: **.MA**

.MA puts all the current conditions of the mouse in various variables. One of those variables is **L2%**, which is 0 until the left mouse button is clicked. (This goes for the joystick fire button as well.) This DotCommand provides the programmer with a tremendous amount of information! Refer to the *Quick Reference Sheet* on the back cover for a complete list of mouse variables that are returned by **.MA.**

UNTIL
Syntax: **.UN,** ***Boolean expression***

.UN is short for *UNtil*, and in the above example the program will loop back to the **.DO** until either **L2%** or PEEK(198) are not 0. So when you click the left mouse button, fire button, or press any key, the program falls through the UNtil. Otherwise, this code just waits for something to happen.

The Do-Loop is perfect for many mouse-driven activities and we will use it a *lot*.

So **GOTO60000** then **RUN** your program! *Always **GOTO60000** before **RUN**ning your program!* That way, if **DB+** is still active in the background, it will be stopped. Running a program with **DB+** still active usually results in an ugly crash. (*OK, "ugly" is too strong a word. But you will have to reset/restart your C64, and your words might be ugly. – Dave*)

You can do all sorts of things with this. For example, a FOR-NEXT loop will add some color and fun:

```
100 for x=0to15:.tx,x:print"Hello, World":next
```

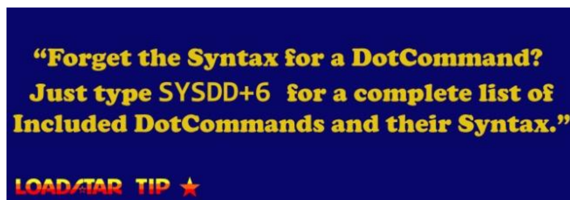
Change line 102 to

```
102 .do:.ma
```

then add:

```
103 .bg,cx%and15
104 .br,cy%and15
```

Now your mouse/joystick will control the color of the background and border. **CX%** gives the text cell X-coordinate (after **.MA** happens), and **CY%** is the Y-coordinate. You will find this feature extremely valuable!



INFINITE COMMANDS: HOW TO "INCLUDE" NEW DOTCOMMANDS WITH DEV

DotBASIC Plus has only a few "built-in" DotCommands, and they include all the ones we learned about in the previous "*Hello, World*" tutorial. In case you skipped it (and shame on you if you did), those DotCommands are:

.tx	Text Color
.bg	Background Color
.br	Border Color
.do	Do Loop
.un	Until
.ma	Mouse Ask
.of	Off (Kill DotBASIC)

As useful as these are, a **BASIC** extension with only a few new commands (eleven, actually—there's also **.WH, .KP, .QS**, and **.QR**) would not be something to get terribly excited about. Fortunately, **DotBASIC Plus** has the ability to add many more. *(100 so far! And the list just keeps growing—Dave)*. However, except for the ones we just mentioned, you will have to "*include*" the new commands in your program. We have to let **DotBASIC** know which DotCommands we will need.

Why go through all this? Why not give me all the commands at once and skip this "including" stuff? Because, since your **DB+** programs will only "include" the DotCommands you need your programs will potentially have a lot more RAM available to them. Why take up valuable memory for dozens and dozens of commands you aren't using? Secondly, the machine language part of your **DotBASIC** programs (that's the file on your Work disk with the .DML extension) only needs to be large enough to contain the DotCommands you choose, saving you a sizable amount of disk space. Finally, and maybe most importantly, this process of "including" makes it very easy for machine language programmers to create new DotCommands that can then be "included" along with the old ones. This way our palette of DotCommands can be expanded into infinity! Imagine the possibilities!

So let's create a new project, one that needs a few new DotCommands we haven't seen yet. With your disks still in the drives, reset your computer, then **LOAD"B.DOTBASIC",*db*** and **RUN**. Choose the drives you are using (as before). This time, for the program name, input **hello2**

After your Work disk is created and the "hello2" program runs, let's **LIST-999** and take a look at the first few lines of code.

```
5 d=peek(186): dd=56*256: mm=16*256
10 rem begin list
20 rem.endlist
30 sysdd
40 .tx,1: print"{clr}";:.bg,6:.br,14
```

Notice lines 10 and 20. To "include" new DotCommands, all we do is enter them between these two lines, in the form of **REM** statements.

Now for the fun. Add this line to your program:

```
11 rem.text,.box,.keymw
```

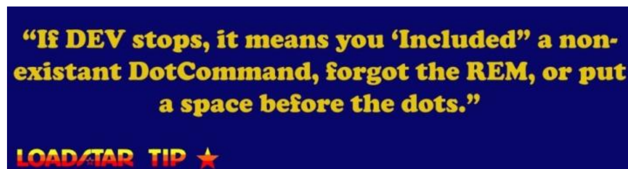
We've "included" three new DotCommands after the **REM** statement, each one separated by a comma. Note that there are no spaces anywhere after the **REM** statement. If you use a DotCommand in your program without "including" it, you will get a **?DOTCOM NOT FOUND ERROR IN LINE xxxx** response.

Save the program (good old **GOTO60000**), then **LOAD"B.DEV",*db*** and **RUN** it. Remember, **LOADSTAR** uses a "b." prefix to denote boot programs. DEV will find your Work disk and present you with a screen very much like the one you see here.

You must run B.DEV after "including" new DotCommands

Choose your **DotBASIC** program (the one that has a .DBS extension) with the **CRSR** keys and press **RETURN**. DEV creates a command list, collects the ML codes necessary, and saves a new HELLO2.DML file to your Work Disk. HELLO2.DML is the ML program that interprets commands from **DB+** and makes them work. The DotCommands "included" between lines 10 and 20 of your **DotBASIC** program are added to this .DML file.

As DEV runs, you will see your included DotCommands listed, along with shared routines and data files. These shared routines are internal **DotBASIC Plus** commands that are used to help create the DotCommands you are adding. It's interesting to see DEV's progress while it is building the new .DML file for your project, but if DEV appears to be adding strange commands like *area, putstr, putint* and so on, don't panic. DEV will give you what you "include" and nothing else. Finally, DEV will **LOAD** and **RUN** your **DotBASIC** program, putting you back in your project (and on the Work Disk). Your "included" DotCommands are ready to go!



How does DEV know which drive your program is on? When you load and run your project, the drive number is tucked away in memory. When DEV is finished, it uses this value to find your program again. So you can use DEV at anytime from any drive to add or remove commands from your program.

*To summarize, you can run DEV as often as you like, anytime you like. Just save your work (**GOTO 60000**) and run DEV from your **DotBASIC Plus** disk. Choose your **DotBASIC** program (with the .DBS extension) and in a few moments you will be right back where you left off—except armed with a few new powerful DotCommands. It's really very simple.*

CREATING TEXT BOXES

Nothing can spruce up the appearance of your program quite like a fancy text box, thus we're going to explore three new DotCommands: **.TEXT, .BOX**, and **.KEYMW**. We've already included them in our **hello2** project from the previous tutorial, so we'll return to that. **LOAD"B.HELLO2",*dw*** then **RUN** and **LIST**. Now let's look at the three commands.

TEXT BOX

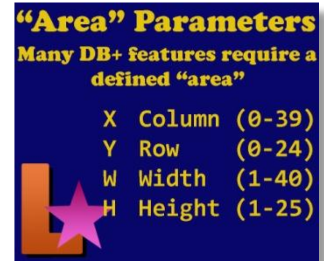
Syntax: **.TEXT,X,Y,W,STRING$**

This command prints the string at **X,Y** — and word wraps it to fit in **W** width. It doesn't get much easier than that! **.TEXT** won't print anything if the text is too long to be wrapped. For example, **.TEXT** will print nothing if **W** equals 10 and **STRING$** = "Antidisestablishmentarianism".

BOX

Syntax:**.BOX,X,Y,W,H,SC,CO**

This is a DotCommand you'll use in nearly every project. **.BOX** draws a box on the screen with the upper left corner at **X,Y**, a width of **W**, and a Height of **H. SC** stands for *Screen Code* — usually 32 for a blank screen or 160 for REVERSEd spaces. **CO** is *COlor*, of course. If you use 255 for **SC**, the area is painted with **CO**. Add 16 to **CO** to draw a frame around the box. This DotCommand does a lot!



KEY/MOUSE WAIT

Syntax:**.KEYMW**

Key/Mouse Wait replaces the Do-Loop we mentioned above. The program stops and waits for either a mouse click or a key press. The mouse variables hold the mouse's current information. **I%** will contain the ASCII number of the key pressed (0 if none).

We are going to draw a box on the screen and fill it with text. First the box:

```
100 .box,5,5,30,07,32,1+16
```

This box has its upper left corner at column 5, row 5, is 30 columns wide and 7 rows high (or deep). The screen code is 32 — a space, with a color of 1 — white. The +16 puts a frame around the box.

Now we need some text to put in the box.

```
110 t$="This is a wonderful test of this remarkable system. "
111 t$=t$+"Nothing can possibly go wrong with this test."
```

T$ now holds this long text. Let's set the text color to yellow, and the border and background to black.

```
112 .tx,7:.br,0:.bg,0
```

We are now ready to print the text with **.TEXT**. Note that the **X,Y** locations are one more than the box's **X,Y**, and the width is two less than that of the box. That's to keep the text from overwriting our frame.

```
120 .text,6,6,28,t$
```

And watch how tidy this is:

```
130 .keymw
```

Be sure to save your program, then **RUN** it.

Whoa! Pretty snazzy for just a few commands. This is a simple example of how seriously powerful **DotBASIC Plus** is.

If you want to do something more, add these lines:

```
140 a$="Clicked Your Mouse"
150 ifi%>0thena$="Pressed the "+chr$(i%) + " Key"
160 .text 6,15,28,a$
170 .keymw
180 print"{clr}":.of:end
```

REGIONS

"Modern Programming" is graphic- oriented, with a GUI — *Graphic User Interface*. With **DotBASIC Plus** we can create some great, "button" controlled programs. The key to creating "buttons" you can mouse over and click on is the concept of *Event Regions*. ***Regions are, in fact, one of the most important concepts in DotBASIC Plus.*** Absorb the following information and you'll be well on your way!

Regions are areas of the screen, defined by you, which the mouse senses when the arrow is over them. These Regions can also be affected by another DotCommand, changing the Region color and other features — much like we've already done with BOXes. So, let's consider three new DotCommands:

DEFINE REGION

Syntax: **.DREG,REG#,X,Y,W,H**

This is sort of like the first part of a **.BOX** DotCommand — just the position of the area on the screen. Each Region has a number, and you can have up to 64 Regions on the screen at a time. So, to define Region #1 we could have a line like this one:

```
100 .dreg,1,10,8,5,3
```

We have defined Region #1. The Region begins at column 10, row 8. The Region is 5 characters wide and 3 characters deep. Simple as that!

Check out **.DREG** in the **DotBIBLE** section to learn a *lot* more about this DotCommand.

AFFECT REGION

Syntax: **.AREG,REG#,SC,CO**

So **.AREG** is like the rest of the BOX command, controlled by the Region number.

PUT MOUSE

Syntax: **.PUTM,X,Y**

This DotCommand puts the mouse arrow anywhere on the screen.

We will also use **.P@, .PC**, and **.BOX** in this program — so it is time to put your disks in their drives, boot up B.DOTBASIC, and create a new program called "buttons".

The first thing to do is add the needed commands to the list:

```
11 rem.p@,.pc,.box
12 rem.dreg,.areg,.putm
```

Save your program with **GOTO60000**, then **LOAD"B.DEV",*db*** from your **DB+** disk, and run it. We are going to change line 40 again — making the background and border both Light Blue (14). You should know how to do that. And we want to use two colors for our buttons — an unhighlighted color and a highlighted color. We will put these in variables to make everything much easier to read.

```
99 un=1:hi=7
```

Now we're ready to get started. The easiest way to position buttons on the screen is to first print the text of the buttons with **.P@**.

PRINT AT

Syntax: **.P@,X,Y,STRING$**

PRINT AT prints **STRING$** at the **X/Y** screen text cell coordinates. *It only prints strings*, so numeric values must be put in **STR$(*value*)**.

A close relative of PRINT AT is PRINT CENTER. We'll be using it in a few moments, and it's even easier.

PRINTCENTER

Syntax: **.PC,Y,STRING$**

This DotCommand prints **STRING$** centered on row **Y**.

Back to our example, we have already chosen locations for four buttons, but feel free to play around with the concept.

```
110 .p@,3,5,"Button One"
120 .p@,20,5,"Button Two"
130 .p@,3,15,"Button Three"
140 .p@,20,15,"Button Four"
```

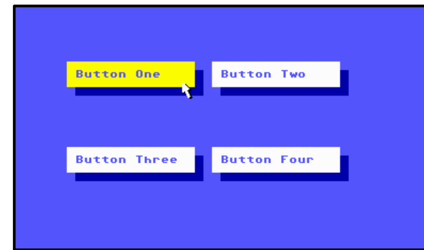
RUN the program to see if this is what you want.

Now we'll create our Regions, defined as areas one space wider than our text in all directions. These Regions will become our clickable "buttons." The **.DREG** DotCommand defines each Region, and on the same line we will use **.AREG** to color in the buttons. Insert these lines:

```
108 .dreg,1,2,4,15,3:.areg,1,160,un
109 .tx,un+128
```

(The **.TX** DotCommand is only necessary on the first button.)

```
118 .dreg,2,19,4,15,3:.areg,2,160,un
128 .dreg,3,2,14,15,3:.areg,3,160,un
138 .dreg,4,19,14,15,3:.areg,4,160,un
```



RUN this. You should have nice white buttons with REVERSE text in each. Now for the shadows. Before the buttons are created, we insert a few lines that create the shaded area. Using BOX, just add 1 in each direction for the **X** and **Y** coordinates, and use a darker color than the background.

```
106 .box,3,5,15,3,160,6
116 .box,20,5,15,3,160,6
126 .box,3,15,15,3,160,6
136 .box,20,15,15,3,160,6
```

Another interesting way to add a shadow effect to your text boxes is to create a BOX with **CO** defined as 255. This will create a box that *shades* the defined area. This creates an especially attractive effect if the area you are shading contains text or graphic characters.

Batta-bing! Your four buttons are ready to go. Our next trick is to place the Mouse Arrow in the lower right corner of Button One.

```
150 .putm,15,6
```

The trick to "*Roll-Overs*" is to keep track of the current button *and* the last button the arrow has pointed at. Since the arrow is now on button 1, we will set a variable (OG) to 1. We also need to change the color of Region 1 to the highlight color.

```
152 og=1
154 .areg,1,255,hi
```

Note that a screen code of 255 in **.AREG** (as with **.BOX**) will paint the color on the existing image.

We have just one other thing to do in preparation for this program — and this is just for classiness! You will understand soon.

```
160 b$(1)="One"
161 b$(2)="Two"
162 b$(3)="Three"
163 b$(4)="Four"
```

Now we are ready for the Do-Loop that will make it all work.

```
200 .do
202 .ma
```

Remember, **.MA** gets mouse information into **DB+**. One piece of information is the Region the arrow is currently over — returned in **RG%**. But if **RG%** = 0 or **RG%** equals the old Region (OG) then we do not want to do anything.

```
210 if rg%=0 or rg%=og then299
```

This line will cause the Do-Loop to loop as long as the Region has not changed and the left mouse button is not clicked. But if the Region has changed, we need to affect the Regions — and update OG.

```
220 .areg,og,255,un:.areg,rg%,255,hi
230 og=rg%
299 .un l2%
```

Take a close look at what is happening, and then **RUN** the program. Of course, at this point, when you click the left mouse button, the program will stop. So we need something a tad more elegant.

```
300 .tx,6
305 .pc,21,"You Pressed Button "+b$(og)
310 .pc,23,"Again (Y/N)"
320 poke198,0
330 .do:.kp,"yn": .un i%
340 if i%=2 then 400
350 .box,1,21,39,3,32,0
360 goto 200
400 .tx,1:.bg,6:.br,14:print"{clr}";
```

In line 305 you now see what those B$(n)s were all about. This is a fairly easy Y/N loop you can use for a myriad of purposes.

In line 330, you'll find another new DotCommand —**.KP**. This is a "built in" DotCommand, so we don't need to include it.

KEYPRESS

Syntax: **`.KP,STRING$`**

This routine quickly scans your string and checks if any of those keys are being pressed at the moment. If one is, that key's position within the string will be returned in **I%**.

MENU MADNESS!

IN THIS SECTION:

Simple Menus 19

DotCommands

BOX 19

STASH 19

RESTORE 19

MENU 19

Multi-Column Menus 22

Scrolling Menus 22

DotCommands:

BLOAD 23

DIRECTORY 23

SCROLLING MENU 23

PROJECT: DOTMENU 24

Mouse Variables 26

Multi-Select Scroll Menus 28

DotCommands

SELECTED ITEM (INDEX) 28

Manual Menu Icons 29

THE HISTORY OF DOTBASIC PLUS

Chuck Peddle and the engineering team of the Motorola 6800 went to work for MOS Technology in 1975 to create the 6502 microprocessor and the KIM 1 test computer. MOS Technology was bought up by Jack Tramiel's Commodore Business Machine company in 1976, and Peddle upgraded the KIM 1 into the PET 2000 – a ready to use home computer.

In 1981, Al Charpentier and Charles Winterble at MOS Tech developed the VIC-II Video Integrated Circuit while Robert Yannes designed the SID – Sound Interface Device – chip. Tramiel called for a home computer built around the capabilities of these two chips in October, and the Commodore 64 was unveiled at the Consumer Electronics Show in Las Vegas in January of 1982.

That same year, Jim and Judi Mangham started a table-top publishing venture called **Softdisk**, creating monthly software collections for the Apple II computer. In 1984, Softdisk, Inc. brought out **LOADSTAR Monthly** for the increasingly popular Commodore 64. Former bar band guitarist Fender Tucker took over Managing Editorship in 1988, and encouraged a generation of independent programmers to greater elegance and style.

At Fender's shoulder was Jeff Jones, a talented programmer who crafted many of **LOADSTAR's** Utility Wares. He also was the ML (Machine Language) guy, and introduced hobbyist programmers to the Toolbox – a collection of ML routines to augment BASIC. Several programmers of the "**LOADSTAR** School" took the idea to further heights, culminating in **Mr. Mouse**, by Lee Novak.

Mr. Mouse had all the features of an ordinary toolbox – menus, boxes, print at, etc. – plus full mouse control. The module went through a number of versions and sizes until Mr. Mouse 2.1 contained nearly everything a programmer might need to create classy point-and-click software.

Dave Moorman became a **LOADSTAR** programmer in 1993, and took to Mr. Mouse immediately. In 2000, he published a PC Windows-ready monthly called eLOADSTAR, which repackaged programs from the 190+ issues of **LOADSTAR**. By the end of the year, when Fender Tucker was ready to set aside **LOADSTAR**, Dave stepped in to continue the "longest running disk magazine in history."

Wanting to create a complete BASIC Extension and software development environment, Dave took Mr. Mouse 2.1 and gave it command names, rather than the SYSaddress commands used in ML modules. The commands all began with a period – and **DotBASIC** was born. The second version, **DotBASIC Plus** – with programmer-customized command lists – is before you now.

Dave sent a copy to Alan Reed, who fell in love with the new language. The documentation, in five C-64 text files, needed some tender loving care – and Alan was just the guy to do it. As he edited the **DB+** Manual, he also worked with the various commands to discover the full capabilities of the language. This, of course, led to fixes and new commands to be added to the library.

Now that you have experienced how easy it is to begin a program in the **DB+** environment, we can go on to some of the powerful commands available to you.

One of the greatest features of **DotBASIC Plus** is its ability to easily create slick, professional looking menus. Drop-down-menus, multi-column menus, file requestors, even multi-select scrolling menus with manually created icons—**DotBASIC Plus** can, as we'll discover, do all these fairly easily using four different menu-creating DotCommands: **.MENU, .MCMENU, .SCMENU,** and **.MSMENU.** We will approach **.MENU** and **.MCMENU** first, since they are the simplest of the menu DotCommands and share certain characteristics. Likewise, **.SCMENU** and **.MSMENU** are slightly more sophisticated and also have features in common.

SIMPLE MENUS

.MENU (MENU) and **.MCMENU** (MULTI-COLUMN MENU) are similar in that the menu choices are created by you and printed to the screen. The menu DotCommands then turn that portion of the screen into a menu. In other words, we set up the screen to look like a menu, then **DotBASIC Plus** turns it into a real menu. Menus like this are very simple to create, and they look great. We will begin by building a program called MENU. MENU, perhaps not surprisingly, uses the **.MENU** DotCommand. While we're at it, we'll try out some other important new DotCommands too.

Our program "MENU" will put a menu on the screen, and then use it to **GOSUB** to line numbers for each item. What the subroutines do is not important — they can do anything you want!

First, put in your Work disk, then **LOAD"B.DOTBASIC",*db*** and **RUN** it. Select your **DotBASIC** drive and Work drive, confirm, and input "**menu**" for the program name. Now you are ready! We will need several new DotCommands added to our .DML file, so don't forget to "include" them and run DEV. Let's take a close look at each of them.

BOX

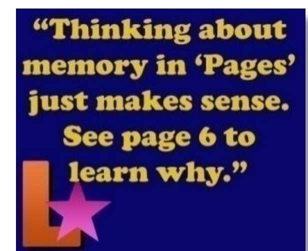
Syntax: **.BOX,X,Y,W,H,SC,CO**

We've seen this one before, but since BOX is such an important DotCommand, we'll talk about it again. BOX puts a box of screen code (**SC**) in color **CO** in the area with the upper left corner at **X/Y**, a Width of **W** (1-40) and a Height of **H** (1-25).

STASH

Syntax: **.STASH,PAGE**

Instantly stashes the whole screen to the given memory **PAGE**. We like Page 208 or 216.



RESTORE

Syntax: **.RESTR,PAGE**

Restores the screen image stashed at given memory **PAGE**. Pages are an easy way to deal with memory. Each page is 256 bytes, so **PAGE***256 is the memory location. You can STASH and RESTORE to memory under ROM and I/O.

And finally…

MENU

Syntax: **.MENU,X,Y,W,I,U,H,HK$**

This DotCommand turns screen rows defined by you into menu lines. **X/Y** set the upper left corner of the menu area. **W** is the *width* of the menu bar, and **I** is the number of *items*. **U** is the color of *unhighlighted* items, and **H** is the color of the *highlighted* item. **HK$** allows us to define "hotkeys" for our menu. More on this later. The number of the menu line chosen is returned in **SL%** (as in ***S**e**L**ection).*

We start by telling **DB+** that we need these DotCommands.

```
11 rem.p@,.pc
12 rem.box,.menu
13 rem.stash,.restr
```

Save your program with **GOTO60000**, then **LOAD"B.DEV",*db*** and **RUN**.

Now, let's get started! In this example, we want to put the menu in a box that has a shadow. This is fairly easy with **DB+**. But first, we will change the screen colors in line 40 to **.BG,0** and **.BR,0**. Now let's put the title of the program on the screen:

```
100 .pc,0,"MY MENU PROGRAM"
```

Next we need a different background. Black does not work well for shadows! We could just change the background color with **.BG,*color***, but that is not fancy enough for us! We want black – the background color – to be the color of *all* our text. Sounds confusing, maybe – but hang in there and all will be made clear. And you'll learn a neat trick to boot!

```
102 .box,0,1,40,18,160,1
```

To parse out this command: the **.BOX** will extend from column 0, row 1 (thus skipping row 0 and our title) for 40 columns and 18 rows. It will use screen code 160 — a REVERSE space — in the color of 1 (white).

Now to put the menu on the screen. This is "in the rough." We just want to find where it will look best.

```
150 .p@,14,5,"A. Item 1{F7}B. Item 2{F7}E. Exit"
```

It so happens we have already positioned this text to be centered under the title. But you can easily move it around by changing the two coordinates (X and Y). The **{F7}**'s are carriage returns to the beginning column on the next line of the PRINT AT. This is a good alternative to using a separate **.P@** for each line.

Now that the menu text is positioned correctly, we need to put a box around it.

```
140 .box,13,4,12,5,160,8
```

Again, we have already done the calculations. But it is pretty easy to figure out. The left edge of the box is one space to the left of the text. The top is one row above the first line of the menu. Width and Height/Items will depend on what is printed.

We are making the box in REVERSE space characters (160) in the color Orange (8). We will want the text to also be in REVERSE orange, so we add a line:

```
141 .tx,8+128
```

*(This is why I like the **.TX** command! Just use the same color value as the box, then add 128 to REVERSE the text—Dave)*

You can run the program at this point if you want. In fact, you can run it after each line is added. And remember, don't forget, do a **GOTO60000** often. *(I do it after adding each line! We have just too many power outages around here!—Dave)*

One last thing— a shadow. List line 140 and change the line number to 138 and press **RETURN**. Now **LIST** 138-140. Edit line 138 by adding 1 to each of the first two values (the upper left corner of the BOX). Then change the color to 15 (light gray).

```
138 .box,14,5,12, 5,160,15
```

When you run the program now, a shadow appears. And you thought shadows were difficult!

Now the Box that is going to contain our menu items is in place and looks suitably snazzy. The items that make up our menu choices are looking good and they are where they should be. The final step is the **.MENU** DotCommand that will turn this into a real menu. Positioning the menu is not hard. Set the **X/Y** to the place where you printed the first line (*"A. Item1"*). Set the **W**idth to one less than the width of the Box. Then set **I** to the number of items in the menu. We'll also add some 'hotkeys' so the user can select the menu items with the keyboard if so desired.

```
160 .menu,14,5,10,3,8,7,"abe"
```

We are using 8 as the unhighlighted color (to match the box) and 7 as the highlighted color. The hotkeys are listed in a string — "abe".

There you go!

We are going to fancy this up just a bit more. We want to have the menu screen come back after a selection is made, so we will Stash it in a nice safe place in memory. The best two pages for screen stashes are 208 and 216 — underneath the I/O.

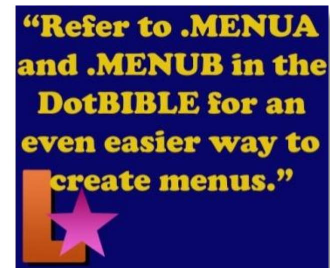
```
155 .stash,208
```

.MENU returns the item number in **SL%**. So all we need to do is create an ON-GOSUB to direct the program. When the subroutine returns, we **.RESTR** the screen we stashed at 208, then GOTO line 160.

```
170 onsl%gosub1000,2000,3000
180 .restr,208
190 goto160
```

Now, we will do something simple for each menu Item.

```
1000 .tx,1:print"{clr}Item 1"
1010 .do:.ma:.un l2%orpeek(198)
1020 poke198,0: return
2000 .tx,3:print"{clr}Item 2"
2010 goto1010
3000 .tx,1:.bg,6:.br,14:?"{clr}"
3010 .of:end
```



You can see how easy it will be to create great programs, listing all the features in the menu. It can also be seen how "modern" looking drop-down menus can be created using **.MENU** along with **.STASH** and **.RESTR. DotBASIC Plus** will take care of the presentation; all you have to do is write the features!

MULTI-COLUMN MENUS

As we've seen, **.MENU** is a pretty powerful DotCommand. But we're just getting started! What if you need a multi-column menu? No problem!

MULTI-COLUMN MENU
Syntax: **.MCMENU,NC,X,W,Y,I,U,HI,HOT$**

This command has a lot in common with **.MENU**, with a few exceptions. **NC** is the number of columns in your menu, which can be from 1 to 5. You must follow that with an **X** coordinate and **W** width value for each column of your menu. **Y** coordinate and **I** (for Items) apply to all columns.

U is the color of unhighlighted items in the menu. The highlight bar is colored **HI**. If you don't want the text to REVERSE or un-REVERSE as the bar moves, add 128 to **HI**.

The user can move the mouse or joystick right to the desired item, or use the **CRSR** keys to change rows and columns. The items are numbered in this order: down the first column, then the next, and so on. So, if you had 3 columns with 7 items in each column, the 2nd column would start with item number 8.

Items can be directly selected by pressing the appropriate *Hotkey* (**HOT$**). The highlight bar is moved to that item number, unless it doesn't exist. Pressing the *Global Escape* key (**MV+12**) ALWAYS returns a zero in **SL%** (*turn to page 26 to learn more about **MV** variables*).

The selected item's number is returned in **SL%**. The maximum number of columns is 5, making the highest possible **SL%** (25 rows) equal to 125. There's a reason for this.

So, a three column menu with three items in each might be created with the following line. We'll shade the **X** and **W** coordinates for each column differently to make it a little clearer:

```
.MCMENU,3,2,8,12,8,22,8,4,3,6,7,HOT$
```

With what you already know about the regular **.MENU** command, try out **.MCMENU**. Consider this a "homework" assignment. Make that "home*play*!"

SCROLLING MENUS

Sometimes your menu has more items than will fit on the screen at one time. A file requestor, for example, is a menu that displays a disk directory. A file requestor can easily have 30, 50, even hundreds of items to choose from. What is needed is a SCROLLING MENU that will give us "Up" and "Down" buttons to click, allowing the user to scroll through the menu selections. **DotBASIC Plus** allows for two such menus, created with two different DotCommands: **.SCMENU** and **.MSMENU.**

Scrolling Menu items aren't simply printed to the screen manually, as with **.MENU.** Instead, Scrolling Menus get their information two ways. First, ***Mr.Edstar*** files saved to your disk can be read in with the BLOAD (**.BL**) DotCommand. ***Mr.Edstar*** is **LOADSTAR's** handy 38 column text editor. ***Mr.Edstar*** files are terminated with a zero. If the file you want to BLOAD does *not* end in zero, use **.BL0** instead. **.BL0** is exactly like **.BL**, with the exception that a zero is tacked onto the end of the BLOADed file.

The second way to get menu items into a scrolling menu is with the **.DIR** DotCommand. **.DIR** loads in a disk directory, then converts it to an ***Mr.Edstar*** format ready to be used by the scrolling menu.

We'll now take a look at these new DotCommands: **.BL,** and **.DIR** to load our menu items into memory; and **.SCMENU** to create a scrolling menu. Then we'll create a useful **DotBASIC** program that shows off some of what we've learned so far.

After walking through a project that uses **.SCMENU** we will take a look at *Mouse Variables*. By **POKE**ing different values into **MV+*offset*** locations, we can customize menus and other **DB+** features.

Finally, we will take a look at the **.MSMENU** DotCommand, that allows us to create a multi-select, scrolling menu.

But first, let's take check out those new DotCommands:

BLOAD

Syntax: **.BL,FILE$,D,LOC**

BLOAD will load files from device **D** (already assigned by **DB+**) to any memory location, except pages 208-223 ($D000-DFFF). This is how we will get your menu items into memory to be used by **.SCMENU**. It can also be used, of course, to BLOAD sprite data, custom character sets, and your machine language routines. **E$** returns the error message. **F%** returns the end location (plus 1) of the BLOADed file. Note that values above 32767 cause the integer **F%** to be negative. Refer to **.I2FP** in the **DotBIBLE** to learn how to convert **F%** to a positive value in these cases.

Remember, **DB+** presets the variable **D** to the current disk drive — just to be handy.

If the file you are BLOADing for a scrolling menu doesn't end with a zero byte, use **.BL0** instead.

DIRECTORY

Syntax: **.DIR,"$:*",D,LOC,#FILENAMES**

This will read the disk directory from device **D** (already defined as the current disk drive number by your **DotBASIC** template program). The directory can be placed anywhere, even under I/O. **DB+** converts the directory to a ***Mr.Edstar*** file as it is brought in. This allows Scrolling Menu to use the information as a file requestor.

Normally you would use **"$:*"** to get all the disk's filenames. You can replace **"$:*"** with any search pattern you want, up to 16 characters long. For example, using **"$:b.*,t.*"** on a **LOADSTAR** disk (using a real C-64 or True Drive in VICE) would bring in the names of all the boot files (those that begin with "b.") and text files (those beginning with "t.").

E$** will return the error message. **T$ will contain the disk's name within quotes, and **B$** will contain the "blocks free" message. Use **VAL(B$)** to extract the number of blocks free on the disk. **N%** returns the number of filenames loaded.

If there was an error during the directory retrieval, **T$** and **B$** will return strings full of spaces, and **E$** will tell about the error.

.DIR has one more parameter: how many filenames you have room to hold. This number should be calculated as:

 # files = INT((bufferspace-1)/32)

For example, if your buffer was from 49152 to 53248 (pages 192-208), you could fit INT((4096-1)/32) = 127 names there. If the buffer space is filled up before all the filenames are loaded, **B$** will return "**more files on disk**". If you don't care about buffer space, use 0 for the number of filenames.

A good place to put your directory information is in pages 224+. You easily put 250 filenames in this area under ROM.

SCROLLING MENU

Syntax: **.SCMENU,X,Y,W,H,B,I,UN,HI,LOC,T$,B$**

For this command to work, you must have the items you wish to menu in memory, and it must end with a zero byte. ***Mr.Edstar*** files end with a zero byte and can be loaded with the **.BL** DotCommand. Otherwise, use **.BL0** to load a file into memory and automatically tack a zero byte on the end. Or you can use **.DIR** to get a disk directory.

The **X,Y,W,H** parameters set the area the menu will occupy (X, Y, Width, Height).

The confusing letters read like this:

Box
Icons
UN-highlight
HIghlight
LOCation

H must be at least 6 characters tall and **W** must be 11 characters or more wide. **B** is the color of the menu box. The unhighlighted items of the menu are color **UN**. **HI** is the highlight bar color. If you don't want the text to REVERSE or un-REVERSE as the bar moves, add 128 to **HI**.

I sets the color of the four words in the corners of the menu: **HOME, UP, DOWN**, and **QUIT**. Left-click on **UP** or **DOWN** to scroll the text. Right-click on them and the list jumps a page at a time.

LOC is the location of the ***Mr.Edstar*** file or directory, which can be anywhere in memory.

The user can use the **CRSR** keys to scroll or page through the text as well.

Clicking on **HOME** will bring the list to the top. Pressing the **HOME** key will bring the highlight bar to the top of the page, and the next press brings the list to the top.

Click on an item or press **RETURN** to select it. The entire item is returned in **W$**. **F$** will return a null unless this is a file requestor - in which case it contains the filename. **SL%** returns the selection number.

Clicking on **QUIT**, pressing **Q**, or pressing the *Global Escape* key (see *Mouse Variables* later in this chapter) will return zero in **SL%** and nulls in **W$** and **F$**.

T$ is printed at the menu's top, in REVERSE, between **HOME** and **UP**. **B$** is printed at the bottom, between **QUIT** and **DOWN**. You needn't use the actual variables **T$** and **B$**, but if this is going to be a file requestor, it's perfect. The user gets to see the disk name and blocks free without any extra effort from you.

Set **W** to 255 and the proper width for a file requestor will be assigned, but that's it. **DB+** knows what filename info looks like and will set **F$** properly when it finds a filename - regardless of width.

Obviously, **.SCMENU** is pretty powerful and has a lot of associated variables and parameters. We will now begin a project that should bring it all together, and when you're finished you'll have a useful program for your software collection.

PROJECT: DOTMENU

DotMENU is an example of the fancy results you can get with just a few lines of code. In fact, this program uses only four different DotCommands.

We now have all the tools we need now to create our first really useful **DB+** program—we'll call it DotMENU. DotMENU will load a disk directory, create a menu of its contents and automatically **LOAD** and **RUN** the program you select. Copy DotMENU and its associated files (B.DOTMENU, MOUSE2.1 7K 1000, DOTMENU.DML, and DOTMENU.DBS) onto a blank disk and you'll have a nice tool to keep your software organized. Even better, you can use **STAR LINKER 2.1** on the **DB+ Utility Disk** to combine the parts of DotMENU into a single file. All this is possible with just a few lines of **DotBASIC** code.

So create a new project called "**dotmenu**" and let's get started!

First we need to 'include" our DotCommands.

```
11 rem.dir,.scmenu,.box,.pc
```

Change the background color to white (1) and the border to m.grey (12) by editing the **.BG** and **.BR** DotCommands in line 40. **GOTO60000** and then run DEV.

The first thing we'll do is use **.BOX** to create a nice looking screen.

```
100 .box,0,3,40,22,160,12
110 .box,0,0,40,3,160,6
120 .box,1,1,38,1,160,14
130 .box,0,24,40,1,160,0
```

We'll display the title for the menu at the top of the screen-- in the center of the blue BOX created by lines 110-120. Also, we'll display a message in the black bar across the bottom of the screen created by line 130.

```
170 .tx,14+128:.pc,1,na$
180 .tx,0+128:.pc,24,"DotMENU 2008"
```

Remember, when setting the text color with **.TX**, add 128 for REVERSE text.

NA$ will represent our disk "title." To make it easy to modify, we'll define this variable at the very top of the program.

```
1 na$="Disk Directory"
```

So, if you put DotMENU on a disk of public domain software, you might define **NA$** as "Public Domain Disk."

We can go ahead and load in the disk directory now.

```
200 .dir,"$:*",d,224*256,250
```

Our directory information is being loaded into page 224 (location 57344), which gives us plenty of room for 250 directory entries.

Now for the menu itself. Note the **W**idth value of 255. Use 255 for **W**idth, and **DB+** will automatically format the menu for a file requestor (or you could just give the menu a **W**idth of 30).

```
220 .scmenu,5,4,255,18,6,0,11,2,224*256,t$,b$
```

The variable **SL%** will hold the number of the menu item selected. If the user selects "Quit" or presses the "Q" key, **SL%** will equal zero, so we need the menu to exit cleanly.

```
230 if sl%=0 then print"{clr}";:.of:end
```

The famous Dynamic Keyboard routine will now **LOAD** the selected program and **RUN** it**.** The name of the file selected will be in **F$**, making the setup a piece of cake.

```
240 print"{clr}{down}{down}{down}loadf$,d"
250 print"{down}{down}{down}{down}run{home}"
260 poke631,13:poke632,13:poke198,2:end
```

It's really that simple. Go ahead and **RUN** it. It's nice, but not quite yet nice enough.

When the **.DIR** DotCommand is executed, the number of files is returned in the **N%** variable. We can use this to give the user a little more information.

```
205 ms$=str$(n%)+” Files on This Disk”:.tx,0+128:.pc,24,ms$
```

Let’s also create a shadow around the directory window. Create a **.BOX** with a color value of 255 to shade the area it affects. It doesn’t matter what we use for **SC** (screen code) in this case, so we’ll use 255.

```
210 .box,6,5,30,18,255,255
```

GOTO60000 and then **RUN**. Impressed? **DotBASIC Plus** can do a lot with very little effort.

There’s one more thing we can do to make it perfect. Did you notice that the menu icons (home, up, down, quit) are all in lower case? We can capitalize these items with two lines of code.

```
150 poke4734,peek(4734)+128:poke4739,peek(4739)+128
160 poke4742,peek(4742)+128:poke4746,peek(4746)+128
```

MOUSE VARIABLES

The Appendix has a complete list of MV values.

MV is a system variable that points to the start of a variable zone—a place where certain user-accessible settings and other data are held. You will use **POKE MV + *OFFSET*** to change how some of the DotCommands work. **MV** is particularly useful in customizing menus, but there are many other cool and interesting features that can be brought to the surface by **POKE**ing new values to **MV**. Let’s highlight just a few of the **DB+** features where **MV** comes into play.

First off, it’s simple to use the keyboard to simulate a left mouse click—just press **RETURN**. But what about right-clicks? The keyboard can mimic the right mouse button too.

MV+14 Right Keycode (F7 = 3)

MV+14 holds the keyboard equivalent to the right mouse button, which is pre-defined as **F7**, but you could change it if you've assigned a function to the right mouse button and would like to use **F7** (and **F8**) for something else.

Note that **MV+14** is not a PET-ASCII code. "*Keycodes*" are generated by the SCNKEY routine during the interrupt. They can be determined with this one-line program:

```
10 print peek(203):goto 10
```

When you **RUN** it, hold down the key you want to designate as a button and note the number showing. Be aware that keycodes are not affected by the special (**SHIFT/C=/CTRL**) keys, and these special keys don't have keycodes - so they can't be used as mouse buttons.

We can make some other choices about how **DB+** interprets keypresses with **MV+18**.

MV+18 Keyboard Enable (default 129)

- **+128** Return can click
- **+64** Space can click
- **+32** C= (Commodore Key) can click.
- **+1** CRSR keys move arrow

Even the **CRSR** keys can control the arrow pointer around the screen. This makes it especially easy to add "keyboard support" to your programs without having to think about it. By default, of course, the **RETURN** key serves as the left mouse button.

With a **POKE**, you could enable the C= key to also serve as the left mouse button. This would be useful for any "click and drag" situations within your program, only because the C= key can be read independently of the **CRSR** keys. It is awkward to use, but it works. **RETURN** or **SPACE** can be used to "click" the rest of the time. It's more natural.

As mentioned earlier**, MV** variables can be used to greatly customize **DotBASIC's** menu DotCommands. Consider what **MV+10**, **MV+11** and **MV+12** can do to change the way menus behave. Note that the numbers in parenthesis are the default values.

MV+10	Menu Type	(192)
MV+11	Multi-Column Menu Type	(192)
MV+12	Global Escape	(0)

MV+12 holds the ASCII code for what you'd like to designate as the escape key for ALL your menus. **SL%** returns a zero when the escape key is pressed.

MV+10 and **MV+11** dictate how your menus will behave. Each bit stands for a specific menu feature. Just add up the values for the features you want and **POKE** that number to **MV+10** or **MV+11**.

MV+11 is used to define features for Multi-Column Menus only. Just like **MV+10**, this byte has eight features attached to it. Only bit 4 (**+16**) is different. **+16** makes Event Regions active and clickable for Multi-Column Menus.

For all other menu types, use **MV+10**. Since **MV+10** and **MV+11** are almost identical, let's deal with them together.

MV+10		**MV+11**	
+128	Automatic Caging of Mouse	**+128**	Automatic Caging of Mouse
+ 64	Automatic Point-to-First	**+ 64**	Automatic Point-to-First
+ 32	Must Select	**+ 32**	Must Select
+ 16	Escape Equal-to-Last	**+ 16**	Click Any Active Region
+ 8	Honor Hotkey Colors	**+ 8**	Honor Hotkey Colors
+ 4	Dual Response	**+ 4**	Dual Response
+ 2	Un-highlight after Select	**+ 2**	Un-highlight after Select
+ 1	Stray-to-Exit	**+ 1**	Stray-to-Exit

+128 causes the mouse to be confined within the menu's borders. After a selection is made, the previous Cage is restored. If auto-caging is not enabled, the user may be able to click on the area outside of the menu, which would return zero in **SL%** (unless it is an active Region). If you *are* using Regions (**MV+11**), you probably shouldn't enable the Cage at all. If you are clicking on an Event Region to flip through "pages" of items, it would be bothersome to have the pointer yanked away with each press.

+64 causes the mouse to be put at the rightmost cell of the first item the instant a regular menu is called. As with auto-Caging (**+128**), enabling this feature might not be a good idea if you are using Event Regions.

+32 causes nothing to happen if the user clicks on the area outside of the menu. The menu remains active as it waits for a real selection. When creating a multi-column menu (**MV+11**), **+32** is especially valuable. Since clicks that occur nowhere can

be ignored, you don't have to make the columns touch. This is made even better if Regions are active. Only Regions and items of the menu could then be clicked on.

+16 causes the escape key, when pressed, to automatically select the last item in your menu - just as if you pressed its hotkey. This saves some code if "Close this Menu" is the last item in your menus - especially when using the **ON-GOTO** command.

> When using **MV+11** and a Multi-Column Menu**, +16** allows the user to click on ANY active Region. If this happens, 128 is added to the Region number and that value is returned in **SL%**. The Region is not "highlighted" in any way as the pointer moves over it. This feature has limited uses and does require some setup on your part. You probably won't want to use the same ol' Regions as the main portion of your program. Special Regions should be defined, making sure that only THEY are the active ones, and labeled in some way to inform the user that these areas are clickable.
>
> If Regions are active, the hotkeys no longer select individual items in the menu, but Regions! 128 will be added to the hotkey's number and that value is returned in **SL%**. Even the fanciest menu can still have complete keyboard compatibility!

+8 causes the highlight bar to not change the colors of characters that have neither the Highlight nor Un-highlight color applied to them. This is an interesting way to inform users of your menu's hotkeys. To use this feature properly, you must make sure the entire menu area is colored in the Un-highlight color, except for the hotkey-characters.

+4 causes the selection number to be returned in **SL%** (as well as the accumulator) when called from ML. This is a ***Mr.Mouse*** feature, and not particularly necessary to **DB+**.

+2 causes the highlight bar to be removed after a selection is made.

+1 causes the menu to be aborted if the mouse strays from the menu area. **SL%** is returned as zero if this happens.

MULTI-SELECT SCROLLING MENUS

If you need to pick multiple items from a menu, **DB+** can accommodate you. This type of menu is by nature a little trickier to setup -- that's why we've saved it for last. The results are worth a little extra effort. We'll need two new DotCommands to make it work.

MULTI-SELECT SCROLL MENU

Syntax**: .MSMENU,X,Y,W,H,B,I,UN,HI,S,WB,LOC,T$,B$**

You can see that **.MSMENU** is not much different from **.SCMENU.** Multi-Select Scroll Menus need two additional parameters, **S** (color of **S**elected items) and **WB** (color of selected item **W**ith menu-**B**ar).

After **.MSMENU** is executed, pressing the **EXIT** key (**MV+16**) or click Quit, **SL%** will return the number of items selected. If **ESCAPE** is pressed or Cancel is clicked (manual icons only), the menu is cancelled and zero is returned in **SL%**. The selected items still exist, and still can be indexed, but you are just told that there weren't any selected items.

So now we know how many items you've selected (**SL%**), how do we access the actual items? For that, we need another DotCommand, **.SEL.**

SELECTED INDEXED ITEM

Syntax: **.SEL,NUMBER**

After using a multi-select menu, **.SEL** is used to ask for each "N"th selected item - whenever and as often as you want - so long as it remains within the current INDEXable file. If that sounds confusing, have no fear—all will be made clear.

Selecting items is as easy as hitting **RETURN** or clicking on them. The item is toggled and the mouse and highlight bar are moved down to the next item, scrolling when necessary - even when selecting with the mouse!

It’s always easy to spot the selected items, even when they are under the highlight bar. Which of the **U,HI,S,WB** items are REVERSEd is set from bits 3-0 to **MV+15**. By default, the highlight bar and all selected items are REVERSEd. Each REVERSE bit can be temporarily disobeyed by adding 128 to the **U,HI,S,WB** parameters.

If you create a Multi-Select Scroll Menu and select 5 items, **SL%** will contain the number 5. The **.SEL** DotCommand will return each item in **W$,** or if the menu consists of a disk directory the filename selected is returned in **F$.** Here is an example:

```
10 for n=1 to sl%
20 .sel,n
30 print w$
40 next n
```

N will increment from 1 to the number of items selected (**SL%**). Each time **.SEL,N** is executed, the **N**th item is returned in **W$** and then displayed.

.MSMENU recognizes these keys in addition to those used in a Scrolling Menu (Up, Down, Home, Quit). With **.MSMENU**, the keys **A, N**, and **T** function to select **ALL, NONE**, and to **TOGGLE ALL** items (respectively). The mouse user cannot access these special features unless you enable manual icons!

If you are using manual icons, the mouse usually can move anywhere on the screen. Nothing happens if the user clicks outside the menu. Even clicking on an active Region has no effect IF the Region number isn't one to which we have assigned a function.

MANUAL ICONS

When defining a Scrolling Menu or a Multi-Select Scrolling Menu, **X+128** enables a powerful new feature. If you do not like the generic look of the scrolling menu, you can now do something about it!

Several things change when **X+128** is used.

1. The **BX,I,T$**, and **B$** parameters are ignored.
2. The scrolling menu is not drawn.
3. The mouse is not caged on the menu.
4. The **X,Y,W,H** parameters now represent the area for the *actual scrolling text* alone.

"*Manual Icons*" means just that. **DB+** is trusting YOU to create, label, and enable (as Regions) the icons for your scrolling menu. They can be any size and at any location.

Here are the Region numbers and the functions they would have for a regular scrolling menu:

1. Home
2. Scroll up
3. Scroll down
4. Exit
5. Page up
6. Page down

Regions 5 and 6 aren't necessary, so don't feel obligated to use them. Nothing bad will happen if you don't have 6 active Regions. Right-clicking on Regions 2 and 3 also page-up and page-down, too. However, joystick users with a repeating fire button may appreciate Regions 5 and 6. No matter what you decide, be sure to include Regions 1-4: the same ones that are present using the generic "standard" icons.

Multi-Select Scroll Menus will obey up to four extra Regions, if you take the time to define them.

7. Select all
8. Select none
9. Toggle all
10. Cancel

Pressing **EXIT** or clicking on Region 4 will return the number of selected items in **SL%**. Region 10 behaves just like **ESCAPE**, returning a grand total of zero selected items.

"When Regions overlap, the higher number has priority. So, if you don't want Regions 5 and 6 in your multi-select menu, just be sure to define Regions 7 and 8 so they perfectly overlap the previous two!

LOADSTAR TIP ★

Since you may not always label the **EXIT** icon as "*Quit*", **MV+16** exists so you can assign an appropriate key to the exit function. The **CRSR** keys still scroll and page, and **HOME** still goes home. **ESCAPE** cancels, just like **EXIT** (in the regular Scrolling Menu).

SOUND & GRAPHICS

IN THIS SECTION:

Bit-Maps and SidPlayer 33

DotCommands

BITMAP 33

SID 33

Grafstar 33

DotCommands:

GRAF 34

MODE 34

PEN COLOR 34

PLOT 34

LINE 34

GET PEN 35

CLIP 35

OFFSET 0,0 35

FILL 35

Scriptor 36

DotCommands

SCRIPTOR 36

SCRIPTOR PRINT 36

THE COMMODORE 64 IS A GRAPHICS MACHINE

BASIC V2, the C-64's built-in language, has a lot of limitations. Its story is well known: when designing the PET, Commodore's first personal computer, founder Jack Tramiel negotiated a deal with Microsoft's Bill Gates for their very popular version of the BASIC programming language. Purportedly, Gates believed the 6502 processor was a toy and he was, in fact, angry when he learned someone on his payroll was working on a 6502 BASIC. When Tramiel and Commodore came along, Microsoft saw an opportunity to cut their losses. They agreed to sell their BASIC to Commodore for a low (some estimates are as low as $10,000) flat-fee. Gates figured, according to one story, that if Commodore was successful they would certainly need revisions and upgrades for future models, thus bringing further revenues to Microsoft. If this is true, Gates certainly did not know Jack Tramiel! Commodore continued to use that version of BASIC, only slightly modified by their own geniuses (they *owned* BASIC 2.0, after all), for years.

When Commodore's new line of home computers, the VIC-20 and C-64, came on the scene BASIC 2.0 was looking very, very long in the tooth. Both these machines had many capabilities that the PETs didn't have at all, but yet the VIC and C-64 were both using what was essentially that original version of PET BASIC. Many new features simply could not be accessed in BASIC at all. On the upside, 20 million or so computers later, Jack Tramiel is one of the few people to ever get the best of Bill Gates in a business deal. (Bill learned a thing or two, and did the same thing when he bought DOS outright.)

Perhaps the greatest let-down in this sad marriage of convenience is BASIC 2.0's complete lack of sound and high resolution graphics commands. Everything, and we mean *everything,* has to be accomplished with PEEKs and POKEs. This is a tolerable (barely) limitation for the VIC-20 since the VIC's simple (but great) sound and graphics features are very straightforward and easy to program, even with nothing but PEEKs and POKEs. Not so with the C-64. Sprites, hi-res graphics, and the incredible SID sound chip are all pretty sophisticated — and maddening to program in BASIC 2.0.

Not anymore! Programming our beloved 8-bit computer is supposed to be a fun mental exercise, not the drudgery of typing two words – PEEK and POKE – over and over again. In fact, many Commodore 64 enthusiasts never really learn how to do sound and graphics programming for exactly this reason. **DotBASIC Plus** changes all that with a truly extensive set of DotCommands that make it all so simple and fun. After all, that's what it's all about, right?

Let's dive right in! We have to be a little more careful about our RAM layout now – some of these new DotCommands eat up pretty significant chunks of available memory. We'll pay close attention to that below, and you can always refer to the Memory Map included on the Quick Reference on the back cover of this book.

BITMAP

Syntax: **.BMP,FILE$,D,160,128,156**
.BMPSCR,SWITCH

SWITCH 1 = Display Bitmap
0 = Display Text Screen

When you include **.BMP** in your program, both **.BMP** and **.BMPSCR** are added to your DotCommand list. Also, an auxiliary file BLOAD is automatically added to your boot program. DBA.UNP.ML is copied to your work disk and loaded into memory — using pages 205-207.

Also, the Top of **BASIC** is lowered from page 160 to 128, to make room for the bitmap, which uses 128-132, 160-191, and 156-159 for display.

In your program, use **.BMP** to load and unpack a **LOADSTAR** .SHP graphic file, using the values listed above. The arrow sprite is automatically copied to page 132 and the screen is switched over by **.BMPSCR,1**.

```
100 .bmp,"graphic.shp",d,160,128,156
110 .bmpscr,1
120 .do:.ma:.un l2%
130 .bmpscr,0
```

Your mouse arrow will not even flicker! Oh, you don't want the arrow on the graphic? Add **POKE 53269,0** to line 110 and **POKE 53269,3** to line 130.

SIDPLAYER

Syntax: **.SID,LOC**
.SIDOFF

DB+ will play **SIDSongs**! When you include .**SID** in your program, **.SID** and **.SIDOFF** are added to your DotCommand List. Also, an auxiliary file BLOAD is added to your boot program so that DBA.SID.ML is loaded into memory — using pages 192-204.

Also, the Top of **BASIC** is lowered from page 160 to 144 (unless **.BMP** is used), to make room for the MUS file(s) at 144-154 (with **.BMP**) or 144-159 (without **.BMP**).

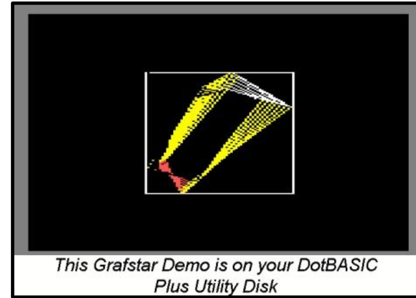
```
100 .bl,"music1.mus",d,144*256
110 .sid,144*256
120 .do:.ma:.un l2%
130 .sidoff
```

GRAFSTAR

The world of bitmap graphics from **DotBASIC Plus** is made possible by a module Dave Moorman wrote years ago — ***Grafstar***. As you can gather from the commands listed below, we have a lot of power here! When you include **.GRAF** in your program, all the other commands listed in the following section are added to your DotCommand List. Also, an auxiliary file BLOAD is added to your boot program so that DBA.GRA.ML is loaded into memory — using pages 134-145. By the way, the ***Grafstar*** DotCommands work just fine along with **.BMP**.

Here are the ***Grafstar*** DotCommands:

```
.GRAF
.GMODE,MODE#
.GPEN,P0,P1,P2,P3
.GPLOT,X,Y,PEN
.GLINE,X,Y,PEN
.GP,X,Y
.GCLIP,X1,X2,Y1,Y2
.GR00,XOFF,YOFF
.GFILL,X,Y,PEN#
```


This Grafstar Demo is on your DotBASIC Plus Utility Disk

Now the details:

GRAF

Syntax: **.GRAF**

This MUST be the first command of the ***Grafstar*** commands used. This initializes the location for the Bitmap — Bitmap at page 160, color at page 128, just like with the **.BMP** command.

MODE

Syntax: **.GMODE,MODE#**

Sets the type of screen that is being displayed.

0 Default Text Screen
1 High Res Bitmap
2 Text Screen — Multi-Color
3 Multi-Color Bitmap
4 Show Text, Clear High Res*
5 High Res Bitmap, Cleared*
6 Show Text, Clear Multi-Color*
7 Multi-Color Bitmap, Cleared*

* Modes 4 - 7 put 0's in all bitmap locations. Modes 4 and 5 also put current colors for pens 0 and 1 into the color map.

If you are using **.GRAF** commands with **.BMP**, use **.BMP,f$,d,160,128,156** to load the SHP graphic, then — rather than **.BMPSCR** — use **.GMODE,1** (hi res) or **.GMODE,3** (multi-color) to switch to the bitmap, and **.GMODE,0** to return to the text screen. *(**.BMPSCR** has a toggling effect that gets cumbersome. Actually, we have fixed the problem, but use **.GMODE** anyway. – Dave)*

PEN COLOR

Syntax: **.GPEN,P0,P1,P2,P3**

This sets the colors of the four "pens" used with **.GPLOT, .GLINE** and **.GFILL**. The four background color registers are used for this, so P0 will change the background of your text screen (use **.BG**,color to reset the background of the text screen after **.GMODE,0**).

PLOT

Syntax: **.GPLOT,X,Y,PEN**

Plots a pixel at pixel **X/Y** in the color set for the **PEN** number. **X/Y** values off the screen are OK, just not visible.

LINE

Syntax: **.GLINE,X,Y,PEN**

Draws a line from the previous **.GPLOT** or the last point of the previous **.GLINE**.

GET PEN
Syntax: **.GP,X,Y**

Returns the pen number of the pixel at **X/Y** in **P%**. If **X/Y** is not visible, **P%** will hold 128.

CLIP
Stntax: **.GCLIP,LEFT,RIGHT,TOP,BOTTOM**

Grafstar uses dynamic clipping to avoid trying to plot a point that is out of bounds. Before calculating the actual memory and bit location, the coordinates are checked against the clip parameters. If not visible, ***Grafstar*** avoids some useless drudgery! But the upshot is that you can set the clip parameters with this command. With **.GCLIP**, you can keep your graphics within a window area.

Note that you must add 1 to the **RIGHT** and **BOTTOM** values. Or to put it another way, **LEFT** and **TOP** is the first point to be plotted while **RIGHT** and **BOTTOM** is the first point to be clipped.

Clip values MUST be within the following range:

LEFT	0 - 319 (High Res)
LEF T	0 - 159 (Multi-Color)
RIGHT	1 - 320 (High Res)
RIGHT	1 - 160 (Multi-Color)
TOP	0 – 199
BOTTOM	1 – 200

Also, **RIGHT** must be greater than **LEFT** and **BOTTOM** greater than **TOP**. ***Grafstar*** checks and produces an **ILLEGAL VALUE ERROR** when something is wrong.

OFFSET 0,0
Syntax: **.GR00,XOFF,YOFF**

Lets you place the coordinates 0,0 ANYWHERE on the screen. Negative coordinates are visible.

XOFF	Number of pixels that the **X** coordinate 0 will be pushed to the right.
YOFF	Number of pixels that the **Y** coordinate 0 will be pushed down.

XOFF and **YOFF** must be legal screen coordinates:

XOFF	0 - 319 (High Res)
XOFF	0 - 159 (Multi-Color)
YOFF	0 – 199

With **OFFSET**, you can define polygons as a series of vertices, in all four quadrants, and position the polygon with the **OFFSET** command. To center the plotting of a polynomial, set the **OFFSET** to the middle of the screen:

```
100 .gr00,160,100
```

Just think of the possibilities!

FILL
Syntax: **.GFILL,X,Y,PEN**

The parameters are identical to PLOT, LINE**,** and GET POINT, and are adjusted in the same way as for OFFSET. FILL will take the pen value at the coordinates given, consider it "empty," and replace every contiguous instance with the **PEN** given in the command, stopping at the occurrence of any location that is “not empty."

You will note that ***Grafstar***'s FILL is reasonably fast. This is because if an entire byte of bitmap memory is "empty," the byte is filled at once: an 8x improvement in plotting speed.

SCRIPTOR

This pair of commands will print text to a bitmap screen. Though primarily for high resolution screens, ***Scriptor*** can put a well designed multi-color font on a multi-color bitmap.

When you Include **.SCRIPT,** you also get .**SCPRNT** for no extra charge – yet another money saving exclusive from **LOADSTAR!**

SCRIPTOR

Syntax: **.SCRIPT,160,128,PAGE** *(usually Page 8)*

You MUST initialize ***Scriptor*** with **.SCRIPT,160,128,LOC**, where **LOC** is the location of the text font. This is normally at page 8, so

```
100 .SCRIPT,160,128,8
```

should work fine. You can **.BL** a font file most anywhere you want.

SCRIPTOR PRINT

Syntax: **.SCPRNT,X,Y,STRING$**

SCRIPTOR PRINT allows you to specify the **X** and **Y** coordinates on the 40 x 25 screen and print your **STRING$** there. You don't place the text at the *pixel* location. This DotCommand prints as if the hi-res screen were a text screen.

Another thing to keep in mind is that the routine prints as if there were a semicolon at the end of the string. If you want to have the cursor act "normally" you'll have to append a carriage return at the end of your string. No problem. **STRING$** becomes **STRING$ +CHR$(13)**.

To print *HOWDY* in the middle of a hi-res screen you'd use:

```
110 .scprnt,17,12,"HOWDY"+chr$(13)
```

That's not all, though. Your string can have almost anything in it, even **CLR, HOME, CRSR UP/DOWN/ RIGHT/LEFT, RVS-ON, RVS-OFF**, and color commands! In hi-res mode you have control of the foreground color and the background color for any character printed.

The routine starts off with any color commands affecting the foreground colors. In other words, the characters themselves.

If you want to change the color of the background of a part of the screen you insert a CTRL-I, or CHR$(9), into your string. It will appear in your code as a REVERSEd "I". To get back to foreground mode, insert a CTRL-H, or CHR$(8) at the beginning of your next string.

You can also get REVERSEd characters the way you always do, inserting a CTRL-9 before your string. CTRL-0 puts you back in regular mode. However, one thing is different.

This routine evolved from **LOADSTAR's** menu system and we don't use normal REVERSEd characters – so it won't do you any good to use a regular 9-block font that has any of the REVERSEd characters customized. All you actually need for ***Scriptor*** are the unREVERSEd characters. ***Scriptor*** will print REVERSEd characters, but they will always be REVERSEd versions of your unREVERSEd characters.

Making ***Scriptor*** work with multi- color bitmaps takes some fiddling around. Since multi-color pixels are double-wide, letters such as M, N, H, and W look a lot alike. However, you can use ***DBDesign*** to design a mulit-color font, using the two alternative colors for "anti-aliasing." The technique is tricky and takes some "thrash and crash" twiddling.

STRINGS AND THINGS

IN THIS SECTION:

Virtual Arrays 39

DotCommands

RACK 39

RACK INDEX 39

PRINT RACKED INDEX 39

Screen Objects 40

DotCommands

SET SOB 40

LINK SOB 40

CUT SOB 40

PASTE SOB 41

DELETE SOB 41

Visual Design 41

DotCommands

FONT/TOOLBOX/STASH 41

SWAP MEMORY 42

Registering DotBASIC Plus 42

TOP DOWN PROGRAMMING – OR HOW TO PLAN BEFORE YOU PLUNGE.

When you think of a particular project – be it a game or something serious – the best thing to do is write down everything you want your code to do. The more you clearly understand where you are going, the better your code will be. Most programs offer a bunch of features which the user can choose at various times. Just make a list!

Then create your Menu! It can be as simple or as fancy as you want. A time-worn layout puts a "menu bar" across the top of the screen, with "File," "Edit," and other groups of functions listed. Each of these words is put in an Event Region. Then wait for one to be clicked:

```
1000 EN=0:.DO
1005 .DO:.MA:.UN CR%
1010 ON CR% GOSUB 2000,3000,4000
1015 .UN EN
1020 .OF:END
2000 .STASH,208: rem stash screen to page 208 to clear the menu later
2006 .BOX,1,1,5,6,255,255: rem shadow box
2008 .BOX,0,0,5,6,160,1:.TX,1+128: rem menu box
2010 .P@,1,1, "Disk{f7}Load{f7}Save{f7}Exit": rem print menu
2020 .MENU,1,1,4,4, "dlse":rem execute menu
2030 .RESTR,208:rem clear the menu by restoring stashed screen
2040 ON SL% GOSUB 2100,2200,2300,2090: rem do features
2050 RETURN: rem return to main loop (1000-1015)
2090 EN=1:RETURN: rem exits program (line 1015-1020)
```

You put your code in each area – 2100, 2200, and 2300. Note how Exit is done – using the variable EN to keep the program looping until it is changed to not zero in line 2090. Then the outer Do-Loop falls through and the program ends.

Do the same for the menus in areas 3000, and 4000 (and as many menus as you need). Now all you have to do is write how each feature is performed.

Should you try your hand at **Visual Basic** or **C++** or some other "Incredibly Big Machine" programming language, you will find that all you have to do is write your Event Handling routines. See – you are already doing it the "professional" way!

You've almost made it! We are now nearly at the end of the Tutorial section of the **DotBASIC Plus** manual. We've covered a lot of territory, and you are probably already programming your Commodore 64 in ways that you never imagined. We've come a long way since **BASIC 2.0**, baby! Believe it or not, though, we've barely scratched the surface of all that **DB+** can do. The **DotBIBLE** section lists many, many more DotCommands that do all sorts of interesting things that should keep you busy for years to come. Thanks to **DotBASIC's** modular design, you can expect even more new DotCommands to come from **LOADSTAR**. Hopefully we'll see new DotCommands created by users like you, too.

In this final chapter of the Tutorial section, we're going to go over a few remaining concepts. Then you're on your own!

VIRTUAL ARRAYS

Never again must you fight for string space in your text adventures and other programs. With the following DotCommands, you can put your lines of text in a ***Mr.Edstar*** file (***Mr.Edstar*** is our nifty text editor that produces 38-column lines of text), use **.BL** or **.BL0** to BLOAD it into memory (under ROM), then use Rack (**.RK**) to turn the text into a *virtual string array*. Rack Index (**.RI**) will put any line into **W$** for use in your program. Let's look at these new DotCommands in depth.

RACK

Syntax: **.RK,LOC**

This routine takes a ***Mr.Edstar*** file (terminated by a zero) that you have BLOADed into memory and "racks it up". By this, we mean that a table of pointers is created right after the zero at the end of the text, enabling you to use the following **.RI** or **.PRI** DotCommands to grab or print individual lines of the file. Thus, we can look at **.RK** as the DotCommand that creates our virtual array.

The file being racked can be located anywhere in memory, even under I/O. Racking needs 3 bytes per line at the end of the file for its pointers. The total number of items in the virtual array is returned in **N%**.

RACK INDEX

Syntax: **.RI,INDEX#**

Once you've racked up a ***Mr.Edstar*** file, you can *index* it. The indexed item is returned in **W$**. **F$** is also set by indexing, and will always return a null unless you happen to be looking at a directory, in which case it contains the entry's filename.

Here is a simple example:

```
100 .bl0,"t.diskover",d,40960
110 .rk,40960
120 for x = 1 TO n%
130 .ri,x
140 print w$
150 next
```

Remember, racking and indexing requires that the file BLOADed be terminated with a zero. ***Mr.Edstar*** files are already terminated with zero, so use **.BL** to BLOAD these. Use **.BL0** for everything else. **.BL0** will BLOAD your file and tack a zero on the end.

PRINT RACK INDEX

Syntax: **.PRI,X,Y,INDEX#**

This routine indexes an item and prints it anywhere on the screen. The string is NOT returned in **W$** or **F$**.

SCREEN OBJECTS: SOBS

Text Cut and Text Paste are DotCommands for copying a portion of the screen into memory, then pasting it anywhere on the screen. However, **.CUT** and **.PASTE** require correctly re-describing the width and height of each cut. Can't we do better? As a matter of fact, we can. With *Screen Objects,* or ***SOBs,*** we've created a group of DotCommands that:

- Set a location in memory for the cut Screen Object.
- Cut to a continuous, indexed stretch of memory, including the width and height information. This memory could then be BSAVEd and BLOADed into a program at will.
- Link a BLOADED *Screen Object Collection* to the program.
- Paste with just the index number and the **X** and **Y** coordinates of the upper left corner of the object.
- Remove Screen Objects from the collection.

Announcing the *SOB* DotCommands, which do just what we want. Now, using **DBDesign**, you can cut portions from any .FTS screen, collect them into an .SOC file, and use the file in any program.

USING SOB COMMANDS

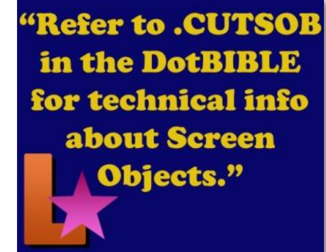
The first thing necessary is telling the program where the SOB Collection is in memory. If this is a new collection (one you are making in the program itself) use the following DotCommand

SET SCREEN OBJECT

Syntax: **.SETSOB,LOC**

where **LOC** is the beginning of the memory where the collection will be. This can be anywhere in memory, even under ROMs and I/O.

If a .SOC file is to be BLOADed, either do the BLOAD after **.SETSOB**, or use



LINK SCREEN OBJECT

Syntax: **.LNKSOB,LOC**

after the BLOAD. Either way will work fine. **.SETSOB** creates an empty collection. **.LNKSOB** just links the memory to the program. **.LNKSOB** also gives us a useful way to keep two or more Screen Object Collection in memory, switching between them when needed.

Cutting a Screen Object is simple:

CUT SCREEN OBJECT

Syntax: **.CUTSOB,X,Y,W,H**

X and **Y** are, of course, the coordinates of the upper left corner of the object, **W** is its width, and **H** is its height. The object is put in the next available space in the collection, and the index total is incremented. Be sure **W** and **H** do not exceed the right or bottom of the screen.

Also, after a cut, the variable **FP** will contain the memory location of the end of the area plus one. This is very handy for saving the collection as a file with BSAVE.

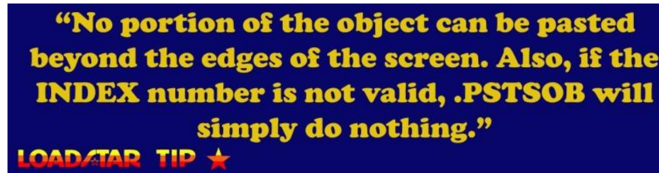
```
100 .setsob,49152
110 .cutsob,0,0,10,10
120 .cutsob,10,0,10,10
130 .cutsob,0,10,10,10
140 .bs,"file.soc",d,49152,fp
```

Now to paste an object to the screen:

PASTE SCREEN OBJECT

Syntax: **.PSTSOB,INDEX#,X,Y**

INDEX# is the number, in order of being cut, of the object, and **X** and **Y** are the coordinates for placing the upper left corner of the object to the screen.



And now for the grand-finale:

DELETE SCREEN OBJECT

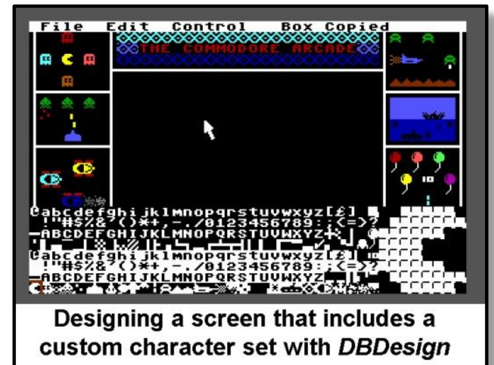
Syntax: **.DELSOB**

.DELSOB simply deletes the last object in memory, which may be useful if memory is limited, or you need to change the last object in the collection.

NOTE: The SOBs use the *current font* for their character patterns.

USING VISUAL DESIGN

The best way to create sharp looking programs is to visually design the screen – your user interface – using ***DBDesign***, which is available on the **DotBASIC** Utility Disk. This program lets you place or type characters on the screen, edit the font, cut and paste sections of the screen, and even cut out Screen Objects and save them to a .SOC file. You can use any of the three text screen modes: Normal, Multi-Color, and Extended Background. And, if you are placing Event Regions on the screen, the ***DBDesign*** BOX function will give you the **X,Y,W,H** of the area – which you can jot down and use for **.DREG**ing your Regions in the program. For complete documentation on ***DBDesign,*** refer to the "Read About It" file on your **DotBASIC** Utility Disk.



Designing a screen that includes a custom character set with *DBDesign*

DBDesign can save your designs in several ways. First, the whole screen, with font and color data, can be saved as an .FTS file which uses 16 memory pages. To put the designed screen in your program, we have a great DotCommand for you:

FONT/TOOLBOX/STASH

Syntax : **.FTS,PAGE**

Using ***DBDesign***, you can visually design a screen, then save it as an .FTS file. This file includes font, screen, color, and text mode information. The file fills 16 pages. Use **.BL** to BLOAD the .FTS file to memory (anywhere except under I/O – `208-223` – we suggest page 224), then use **.FTS** to put everything on the screen. Instantly!

```
100 .bl,"myprog.fts",d,224*256
110 .fts,224
```

Instantly, your pre-designed screen is staring back at you!

The second way ***DBDesign*** can save your screen and color information only as a .TBS file. TBS stands for *Tool-Box-Stash*. .TBS files are good to use if multiple screens share the same font. To display a .TBS file, simply BLOAD it into RAM and use **.RESTR** to put it on the screen, like this:

```
1000 .bl,"screen.tbs",d,160*256
1002 .restr,160
```

Finally, ***DBDesign*** can save the font only. This allows you to easily change your character set in a program. **DotBASIC Plus** uses Page 8 for character sets (even if it *appears* you are using the default character set), and a complete character set fills 8 pages. Changing your font, then, is as simple as BLOADing it into Page 8, like this:

```
1000 .bl,"f.screen",d,8*256
```

Shazzam! That was easy! What if you have two fonts, and you want to keep them both in RAM, swapping them in and out when needed? No problem! To do it, we need another DotCommand. You'll love this one!

SWAP MEMORY

Syntax: **`.SWPMEM,START,END+1,DESTINATION`**

SWAP MEMORY will swap the area of memory with that at the **DESTINATION**, at a rate of 41 cycles per byte. (Having the destination between the start and end is NOT a good idea with **.SWPMEM**!)

To use SWAP MEMORY to swap our two fonts in and out:

```
100 .bl,”f.font1”,d,8*256               (Displays Font #1)
110 .bl,”f.font2”,d,224*256             (BLOAD Font #2 into Page 224)
120 .swpmem,8*256,16*256,224*256        (Swap!)
```

To swap back, just issue the **.SWPMEM** again, exactly like before.

"Don't miss new and improved DotCommands for DotBASIC Plus. Check in often at the DotBASIC Forum for news, commentary, and upgrades. The forum is also the place to go for DotBASIC help and advice.

http://8bitcentral.com/dotbasic

The DotBIBLE

DotBasic Infinite Burgeoning Language Explanation

SHORT, MEMORABLE, CONCISE, AND OPAQUE

Incredible synergy comes when two writers/editors merge to create the ultimate manual for the ultimate language. However, Dave and Alan had their "moment:"

Dave's version: *When it came to a name for the complete listing of DotCommands, Alan and I hit an impasse. He wanted to use the term "bible." I, being a clergy in my real life, had problems with that term. We might have come to blows were it not for the fact that we live some 1000 miles apart.*

Alan's version: *Dave, being a preacher, wanted to use the term "bible." I was not so sure the term was appropriate, and thought "Command Summary" was more fitting. Fortunately, I live in Ohio and Dave is in Colorado. Also, I am younger and more agile than Dave.*

On the other hand, both Alan and Dave agreed that the best names in computerdom are short, memorable, concise, and opaque — like PET, which stands for **P**ersonal **El**ectronic **T**ransactor. Alan suggested the name for the Summary be "Dot-something." Dave tossed around some random acronymic possibilities, and suggested **Dot B**asic **I**nfinite **B**urgeoning **L**anguage **E**xplanation.

DotBIBLE!

Call it what you may —the following is the official list of DotCommands as of 2008.

.ALPH ALPHABETIZE

Syntax: **.ALPH,START#**

DESCRIPTION: Sorts racked data, treating the **START#** character in each string as the first character sorted. Normally, **START#** would equal one (1), of course. However, changing **START#** can be useful if you need to sort a directory of **LOADSTAR** text files, for example, and want to ignore the "T." prefix on each file.

See also: .DIRSRT, .RK

.AREG AFFECT REGION

Syntax: **.AREG,REG#,SC,CO**

SC=255	**PAINT**
CO+16	**BLOCK**
CO+32	**FRAME**
CO+64	**UN-REVERSE***
CO+128	**REVERSE***
CO+192	**FLIP***
CO=255	**SHADE**

**Color RAM will not be affected unless you add an additional 16 to the values above.*

DESCRIPTION: Affect the specified Region with **SC** (Screen Code) and **CO** (Color). This command is very similar to **.BOX**, but instead of specifying the **X,Y,W,H** parameters, you indicate which defined Region (using **.DREG**) you want to affect.

See also: .BOX, .DREG, .DRTEXT, .EDRTEXT, .ROLOVR

.BG BACKGROUND COLOR BUILT-IN

Syntax: **.BG,CO**

DESCRIPTION: Change the background color to the value of **CO**.

See also: .BR, .TX

.BL BLOAD

Syntax: **.BL,FILE$,D,LOC**

DESCRIPTION: Performs a binary load from device **D** to any memory location, except Pages 208 - 223.

Related Variables:

E$	Returns the error message.
F%	Returns the end location (plus 1) of the BLOADed file. Values above 32767 cause **F%** to be negative (use **.I2FP** to convert **F%** to a positive number).

See also: .BL0, .BS, .DIR, .DISK, .I2FP, .RK

.BL0 BLOAD WITH ZERO

Syntax: **.BL0,FILE$,D,LOC**

DESCRIPTION: Performs a binary load from device **D** to any memory location, except Pages 208 - 223, and adds a zero (0) to the end of the LOAD. This can be useful when using Scrolling Menus and other DotCommands that expect their data to end with a zero byte.

Related Variables:

E$	Returns the error message.
F%	Returns the end location (plus 1) of the BLOADed file. Values above 32767 cause **F%** to be negative (use **.I2FP** to convert **F%** to a positive number).

See also: .BL, .BS, .DIR, .DISK, I2FP, .RK

.BMP LOAD SHP FILE

Syntax: **.BMP,FILE.SHP$,D,160,128,156**

DESCRIPTION: Used with the **.BMPSCR** DotCommand, **.BMP** loads and unpacks a **LOADSTAR** SHP graphic file, using the values listed above. The arrow sprite is automatically copied to page 132 and switched over by **.BMP**. When you include **.BMP** in your program, both **.BMP** and **.BMPSCR** are added to your DotCommand list.

An auxiliary file BLOAD is added to your boot program so that DBA.UNP.ML is copied to your Work disk and loaded into memory — using pages 205-207.

Also, the Top of **BASIC** is lowered from page 160 to 128, to make room for the bitmap, which uses 128-132, 160-191, and 156-159 for display.

See also: .BMPSCR

.BMPSCR DISPLAY SHP FILE INCLUDED WITH .BMP

Syntax: **.BMPSCR,SWITCH**

SWITCH **1 = Display Bitmap**
0 = Display Text Screen

DESCRIPTION: After loading a SHP bitmap with **.BMP**, use **.BMPSCR** to either display the SHP file or return to the text screen. **POKE 53269,0** to turn off the mouse pointer and **POKE 53269,3** to restore it.

EXAMPLE:

```
100 .bmp,"graphic.shp",d,160,128,156
110 .bmpscr,1
120 .do:.ma:.un 12%
130 .bmpscr,0
```

See also: .BMP

.BOX BOX

Syntax: **.BOX,X,Y,W,H,SC,CO**

SC=255	PAINT
CO+16	BLOCK
CO+32	FRAME
CO+64	UN-REVERSE*
CO+128	REVERSE*
CO+192	FLIP*
CO=255	SHADE

**Color RAM will not be affected unless you add an additional 16 to the values above.*

DESCRIPTION: Draws a box made up of the **SC** Screen Code, in the color defined by **CO**. The box is drawn on the screen with the upper left corner at **X,Y**, a width of **W**, and a Height of **H.**

See also: .AREG, .FANCY, .TEXT, .TEXTC

.BR BORDER COLOR BUILT-IN

Syntax: **.BR,CO**

DESCRIPTION: Changes the border color to the value of **CO**.

See also: .BG, .TX

.BS BSAVE

Syntax: **.BS,FILE$,D,START,END+1**

DESCRIPTION: Save a section of memory to **D**. Only memory accessible by the CPU may be saved - meaning you can't save data hidden under the ROMS or I/O.

<u>Related Variables:</u>
E$ Returns the error message.
See also: .BL, .BL0

.CAGEM CAGE MOUSE

Syntax: **.CAGEM,X,Y,W,H**

DESCRIPTION: Confines the mouse pointer's movement within an area.

See also: .LG, .PUTM

.CHRSWP CHARACTER SWAP

Syntax: **.CHRSWP,SEEK,REPLACE,CO**

***DESCRIPTION**:* Scans the screen for a specific screen code (**SEEK**), and replaces it with the given screen code (**REPLACE**), with the given color (**CO**). A color of 128 causes only the characters to be changed, not their colors.

See also: .COLSWP

.COLSWP COLOR SWAP

Syntax: **.COLSWP,SEEK,REPLACE**

DESCRIPTION: This routine finds all instances of a specific color (**SEEK**) and replaces them with the given color (**REPLACE**). Screen memory is not affected.

See also: .CHRSWP

.CPYCHR COPY CHARACTER

Syntax: **.CPYCHR,START,END+1,DESTINATION**

DESCRIPTION: Lifts the ROMs and exposes the Character-ROM in the $D000 area so you can copy it.

See also: .CPYMEM, .SWPMEM, .CPYIO

.CPYIO COPY I/O INTACT

Syntax: **.CPYIO,START,END+1,DEST**

DESCRIPTION: Lift the ROMs, but leave the $D000 area alone. This would be useful in copying the color RAM or the VIC-II's settings.

See also: .CPYCHR, .CPYMEM, .SWPMEM

.CPYMEM COPY MEMORY

Syntax: **.CPYMEM,START,END+1,DEST**

DESCRIPTION: All ROMS are lifted and a raw memory transfer is performed at a speed of approximately 28 cycles per byte. If the **DEST**inaton lies somewhere between the **START** and **END**, a backwards copy is performed to prevent corruption.

See also: .CPYCHR, .CPYIO, .SWPMEM

.CUT CUT

Syntax: .CUT,X,Y,W,H,LOC

DESCRIPTION: Stash any size portion of the screen to any location, even under I/O. The data is stored sequentially in memory with each cell's screen code and color code stored one after another.

To determine how much memory is consumed by **.CUT**, use this formula:

bytes = W * H * 2

See also: .PASTE, .CUTSOB, .PSTSOB

.CUTSOB CUT SCREEN OBJECT

Syntax: **.CUTSOB,X,Y,W,H**

DESCRIPTION: The screen object defined by **X,Y,W,H** is put in the next available space in the Screen Object Collection, and the index total is incremented. Be sure **W** and **H** do not exceed the right or bottom of the screen.

After using **.CUTSOB**, the variable **FP** will contain the memory location of the end of the area plus one.

The Screen Object data is arranged as follows:

BYTE	
0	Index number
1-2	Offset to end of collection
3-4	Offset to next screen object
5	Width
6	Height
7	Screen Object data alternating Screen Code/Color bytes

.CUTSOB uses three ML DotCommands:

.AREA which turns our **X,Y,W,H** to **X1,X2,Y1,Y2** format (for compatability with **Mr.Mouse**)
.MULTIPLY which multiplies .A and .X and puts the results in 834/835
.PUTIFP which puts a two-byte interger value in .A/.X into a floating point variable, from the name given in 251/252

We also use **#SOBDATA** as a data block containing the beginning address of the collection, used by all four commands.

Related Variables:

FP End of SOC+1

See also: .CUT, .DELSOB, .LNKSOB, .PASTE, .PSTSOB, .SETSOB

.DELSOB DELETE SCREEN OBJECT

Syntax: **.DELSOB**

DESCRIPTION: Deletes the last Screen Object in memory.

See also: .CUT, .CUTSOB, .LNKSOB, .PASTE, .PSTSOB, .SETSOB

.DISK DISK COMMAND

Syntax: **.DISK,COMMAND$,D**

DESCRIPTION: Sends string (**COMMAND$**) to device **D** via the command channel. To only read the error channel, send a null string (""). **COMMAND$** cannot be a variable; it must be a literal string.

Related Variables:

E$ Returns the error message.

.DIR GET DIRECTORY

Syntax: **.DIR,"$:*",D,LOC,#FILENAMES**

DESCRIPTION: Reads the disk directory from device **D** and stores it in **LOC**. The directory can be placed anywhere, even under I/O. **DB+** converts the directory to an ***EDSTAR*** file as it is brought in. This allows Scrolling Menu to use the information as a file requestor. You can replace **"$:*"** with any search pattern you want, up to 16 characters long. For example, using **"$:b.*,p.*"** on a **LOADSTAR** disk would bring in the names of all the boot files (those that begin with “b.” and packed files (those beginning with “p.”).

The number of filenames a given buffer area can hold can be determined by:

 # files = INT((bufferspace-1)/32)

A good place to put your directory information is in pages 224+. You easily put 250 filenames in this area under ROM.

Related Variables:

E$ Returns the error message.
T$ Returns the disk header information, in quotes.
B$ "Blocks Free" message.

See also: .DIRSRT, .PSEL, .SCMENU

.DIRSRT SORT DIRECTORY

Syntax: **.DIRSRT**

DESCRIPTION: Alphabetically sort an already BLOADed and racked directory.

See also: .ALPH, .DIR

.DO DO BUILT-IN

Syntax: **.DO:*loop***

DESCRIPTION: Begin a Do Loop.

See also: .MA, .UN, .WH

.DREG DEFINE REGION

Syntax: **.DREG,REG#,X,Y,W,H**

DESCRIPTION: Any area of the screen can be defined as a "Region". You specify the Region number (1-64) and its area. When Regions overlap each other, the highest numbered Region prevails.

Related variables		Defaults:
MV+0	Region Data Zone: LB	(0)
MV+1	Region Data Zone: HB	(45)
MV+2	# of Active Regions	(0)

MV+1 normally has a value of 45, which assigns the Region data to page 45.

Defined Regions are not "seen" by the mouse unless you mark them as *active*, by placing a value into **MV+2**. For example, if this number is 7, then Regions 1-7 are active. Each time you define a Region, that Region number is automatically placed into **MV+2** for your convenience. That means if you have defined three Regions, than redefine Region 2, you will need to **POKE MV+2,3** to restore all the Regions.

Region data can be placed almost anywhere, the exception being under the ROMS or I/O. Pages 45-55 are available for this use in **DB+**. (This is also the area for Sprite Images.) If you want to change the location of your Region data, you should first set the Region Data Zone with **POKE**s to **MV+1** and **MV+0** before defining Regions. So, if you change the location of your Region data, it's your job to ensure this area is safe from **BASIC** and your other data.

See also: .AREG, .BOX, .DRTEXT, .EDRTEXT, .ROLOVR

.DRTEXT DEFINE REGION TEXT

Syntax: **.DRTEXT,NUMBER,"STATIC STRING"**

DESCRIPTION: "*Region text*" is when the user moves the mouse pointer around the screen, and a message bar at the bottom of the screen informs the user of what will happen if he or she clicks on that particular area. These strings don't have to be associated with Regions - it's just likely that this will be their most common use.

Strings defined as Region text must NOT be made by combining smaller strings. The string's POINTER will be stored in its proper slot in the Region Text Zone. Be sure this zone is safe from **BASIC** and other data. We suggest the area in pages 46-55. The Zone will never exceed 3 pages.

All Region text will be printed in **MV+22's** color. Add **128** to **MV+22** for REVERSE printing. (Adding **64** changes the way the pointers are stored and is most useful from ML.)

Add **32** to **MV+22** and all your Region text will be **CENTERED**.

Add **16** instead, and each string will be printed after a forced leading space.

Related Variables:		Defaults:
MV+20	Region Text Zone (LB)	(0)
MV+21	Region Text Zone (HB)	(4)
MV+22	Region Text Color / Flags	(1)
MV+23	Region Text Row	(24)

See also: .AREG, .BOX, .DREG, .EDRTEXT, .PRTEXT, .ROLOVR

.EDRTEXT EDSTAR TO REGION TEXT

Syntax: **.EDRTEXT,LOC**

DESCRIPTION: This command defines ALL Region text with a single command! It takes an ***EDSTAR*** file (terminated by 0), racks it up, and POKEs **LOC** to **MV+24** and **MV+25**. The number of lines in the file is returned in **N%**.

Keep in mind that the FIRST line of the ***EDSTAR*** file will be referenced by number zero. You can have as many lines as you want.

See also: .AREG, .BOX, .DREG, .EDRTEXT, .PRTEXT, .ROLOVR

.EVENT EVENT

Syntax: **.EVENT,"*keystroke*"**

DESCRIPTION: Waits until Mouse Left or Right button is clicked, or Key is pressed and character is in "keystroke". If Mouse click**, I%** = -1 (Left) or -2 (Right). All mouse variables are current. If keystroke**, I%** = position in "keystroke". The keystroke string can be literal or variable.

This handy routine also does *Roll-Overs* so it's important to set them up with **.SETROL** before calling **.EVENT**.

See also: .DO, .MA

.FANCY FANCY LATTICE

Syntax: **.FANCY,X,Y,W,H,S1,S2,C1,C2**

DESCRIPTION: Draws an alternating pattern from the two screen codes (**S1/S2**) and colors (**C1/C2**) you specify.

See Also: .BOX

.F2SPR FONT-TO-SPRITE

Syntax: **.F2SPR,SC,SPRITEIMAGE#**

DESCRIPTION: Copies font character of **SC** (screen code) to Sprite Image (184-219)

See also: .SPRITE, .SPRFX, .SPRMV

.FTS FONT/TOOLBOX/STASH

Syntax: **.FTS,PAGE**

DESCRIPTION: Using ***DBDesign***, you can visually design a screen, then save it as an FTS file. This file includes font, screen, color, and text mode information. The file fills 16 pages. Use **.BL** to BLOAD the FTS file to memory (anywhere except under I/O — 208-223 — we suggest page 224), then use **.FTS** to put everything on the screen. Instantly!

EXAMPLE:

```
100 .bl,"file.fts",d,224*256
110 .fts,224
```

Note: **.BL0** should NOT be used for .FTS files. (that extra 0) will fall into the next Page of memory.

See also: .BL, .BL0

.GRAF GRAFSTAR

Syntax: **.GRAF**

DESCRIPTION: Initializes the location for the Bitmap — Bitmap at page 160, color at page 128, just like with the **.BMP** command. Including **.GRAF** and running DEV also includes the rest of the ***Grafstar*** suite of DotCommands: **.GMODE, .GPEN, .GPLOT, .GLINE, .GP, .GCLIP, .GR00, and .GFILL**

See also: .GCLIP, .GFILL, .GLINE, .GMODE, .GPEN, .GPLOT, .GP, .GR00

.GCLIP CLIP INCLUDED WITH .GRAF

Syntax: **.GCLIP,X1,X2,Y1,Y2**

DESCRIPTION: Restricts ***Grqfstar*** graphics within a window area. Note that you must add 1 to the **X2** (RIGHT) and **Y2** (BOTTOM) values. Or to put it another way, **X1** and **Y1** is the first point to be plotted while **X2** and **Y2** is the first point to be clipped.

Clip values MUST be within the following range:

```
X1 (LEFT)     0 - 319 (High Res)
X1 (LEFT)     0 - 159 (Multi-Color)
X2 (RIGHT)    1 - 320 (High Res)
X2 (RIGHT)    1 - 160 (Multi-Color)
Y1 (TOP)      0 – 199
Y2 (BOTTOM)   1 – 200
```

Also, **X2** must be greater than **X1** and **Y2** greater than **Y1**. ***Grafstar*** checks and produces an **?ILLEGAL VALUE ERROR** when something is wrong.

See also: .GFILL, .GLINE, .GMODE, .GPEN, .GPLOT, .GP, .GR00, .GRAF

.GFILL FILL INCLUDED WITH .GRAF

Syntax: **.GFILL,X,Y,PEN#**

DESCRIPTION: Takes the pen value at the coordinates given, considers it "empty," and replaces every contiguous instance with the **PEN#** given in the command, stopping at the occurrence of any location that is “not empty."

See also: .GCLIP, .GLINE, .GMODE, .GPEN, .GPLOT, .GP, .GR00, .GRAF

.GLINE LINE INCLUDED WITH .GRAF

Syntax: **.GLINE,X,Y,PEN**

DESCRIPTION: Draws a line from the previous **.GPLOT** or the last point of the previous **.GLINE**.

See also: .GCLIP, .GFILL, .GMODE, .GPEN, .GPLOT, .GP, .GR00, .GRAF

.GMODE MODE INCLUDED WITH .GRAF

Syntax: **.GMODE,MODE#**

DESCRIPTION: Sets the type of screen that is being displayed.

Modes:
0 Default Text Screen
1 High Res Bitmap
2 Text Screen — Multi-Color
3 Multi-Color Bitmap
4 Show Text, Clear High Res*
5 High Res Bitmap, Cleared*
6 Show Text, Clear Multi-C*
7 Multi-Color Bitmap, Cleared*

* Modes 4 - 7 put 0's in all bitmap locations. Modes 4 and 5 also put current colors for pens 0 and 1 into the color map.

If you are using **.GRAF** commands with **.BMP**, use **.BMP,F$,D,160,128,156** to load the SHP graphic, then — rather than **.BMPSCR** — use **.GMODE,1** (hi res) or **.GMODE,3** (multi-color) to switch to the bitmap, and **.GMODE,0** to return to the text screen. (**.BMPSCR** has a toggling effect that gets cumbersome.)

See also: .GCLIP, .GFILL, .GLINE, .GPEN, .GPLOT, .GP, .GR00, .GRAF

.GP GET PEN INCLUDED WITH .GRAF

Syntax: **.GP,X,Y**

DESCRIPTION: Returns the pen number of the pixel at **X/Y** in **P%**. If **X/Y** is not visible, **P%** will hold 128.

Related Variable:
P% Pen #

See also: .GCLIP, .GFILL, .GLINE, .GMODE, .GPEN, .GPLOT, .GR00, .GRAF

.GPEN DEFINE PEN COLOR INCLUDED WITH .GRAF

Syntax: **.GPEN,P0,P1,P2,P3**

DESCRIPTION: This sets the colors of the four "pens" used with **.GPLOT, .GLINE** and **.GFILL**. The four background color registers are used for this, so **P0** will change the background of your text screen (use **.BG**,color to reset the background of the text screen after **.GMODE,0**).

See also: .GCLIP, .GFILL, .GLINE, .GMODE, .GP, .GPLOT, .GR00, .GRAF

.GPLOT PLOT INCLUDED WITH .GRAF

Syntax: .**GPLOT,X,Y,PEN**

DESCRIPTION: Plots a pixel at pixel **X/Y** in the color set for the **PEN** number. **X/Y** values off the screen are invisible.

See also: .GCLIP, .GFILL, .GLINE, .GMODE, .GP, .GPEN, .GR00, .GRAF

.GR00 SET OFFSET INCLUDED WITH .GRAF

Syntax: **.GR00,XOFF,YOFF**

DESCRIPTION: Lets you place the coordinates 0,0 ANYWHERE on the screen. Negative coordinates are visible.

XOFF Number of pixels that the **X** coordinate 0 will be pushed to the right.
YOFF Number of pixels that the **Y** coordinate 0 will be pushed down.

With **.GR00**, you can define polygons as a series of vertices, in all four quadrants, and position the polygon with **.GR00**. To center the plotting of a polynomial, set the offset to the middle of the screen:

```
.GR00, 160, 100
```

See also: .GCLIP, .GFILL, .GLINE, .GMODE, .GP, .GPEN, .GPLOT, .GRAF

.I2FP INTEGER-TO-FLOATING POINT

Syntax: **.I2FP,INTEGER**

DESCRIPTION: Some **DotBASIC** variables return negative values for numbers greater than 32767. BASIC 2.0's FRE(0) command is another example. The unsigned value is returned in the variable **FP**.

Related Variables
FP Unsigned value of **INTEGER**

EXAMPLES:

```
.i2fp,fre(0):print fp
.i2fp,f%:print fp    (after a BLOAD)
```

See also: .BL, .BL0

.INP TEXT INPUT

Syntax: **.INP,X,Y,TXT,CSR,LEN,DEFAULT$**

X+128 REVERSE input
Y+128 Integer numbers only
Y+192 Positive and negative decimal numbers only

DESCRIPTION: When called, **DEFAULT$** is printed at **X,Y** followed by a blinking cursor. The user can **CRSR** through the text, **INSERT, DELETE, HOME**, or **CLR** at will. Most normal characters are allowed, except for those fearsome quotation marks. The string is returned in **W$**.

TXT is the text color and **CSR** is the cursor color. **LEN** is the maximum number of characters allowed, which cannot exceed 80. The default string (**DEFAULT$**) will be cut short if it exceeds **LEN**.

A nice thing about **.INP** is that it automatically clears out the space it needs, which is handy for inputting over fields that might contain old data.

To input in REVERSE, add 128 to **X**. If you want to allow only numbers to be entered, add 128 to **Y**. If you want to allow numbers AND the decimal and minus symbols, add 192 to **Y**.

The *Keyboard Enable* variable at **MV+20** is temporarily zeroed during the **INPUT** routine. It looks silly if the arrow pointer moves back and forth as you are **CRSR**ing through the text.

Related Variables:
W$ Result of user input

.INPLUS INPUT PLUS

Syntax: **.INPLUS,X,Y,TXT,CSR,LEN,S,OUT$,DEFAULT$**

X+128 REVERSE input
Y+128 Integer numbers only
Y+192 Positive and negative decimal numbers only

DESCRIPTION: When called, **DEFAULT$** is printed at **X,Y. TXT** is the text color and **CSR** is the cursor color. **L** is the maximum length allowed, which cannot exceed 80. The user can **CRSR** through the text, **HOME, CLR, INSERT**, and **DELETE** at will.

If you want to input in REVERSE, add 128 to **X**. To allow only numbers to be entered, add 128 to **Y**. To allow numbers AND the decimal and minus symbols, add 192 to **Y**.

S is the starting location of the blinking cursor. For example, if **S** was 7, then the cursor would pop up over the 8th character of the default string. *Why not the 7th?* Well, zero is used to represent the leftmost (1st) character. If you positioned your **.INPLUS** against the left border, you'd see that **S** corresponds to the usual 0-39 Cell-X values.

If **S** exceeds the length of the default string (**DEFAULT$**), the cursor would pop up right after the end of **DEFAULT$**. Since this string can never exceed 80 characters, any **S** value above 80 ensures the cursor will start off at its "normal" position at the end of **DEFAULT$**.

OUT$ allows you to specify extra keys you want to act like **RETURN**. For example, you might have a list of figures and would like to be able to use **CRSR UP/DOWN** to move through each one, and then press **F1** to signify when you're done altering the lot.

The string is returned in **W$**. **I%** tells you WHICH exit-key was pressed. It is zero when **RETURN** is used, and any other value means that specific key from **OUT$** was pressed to exit.

Since the incoming keypresses are checked against **OUT$** first, you could use **CRSR LEFT, RIGHT, HOME**, or ANY key to act as a special **RETURN** key. Of course, using **LEFT** and **RIGHT** as exit keys means they couldn't be used to move the blinking cursor within the current line of input. However, if you had a bunch of values in rows and columns, using all four **CRSR** keys to move through items would be handy!

Related Variables:
W$ Result of user input
I% “Exit” key

.INSTR IN-STRING

Syntax: **.INSTR,SEARCH$,TARGET$,N**

DESCRIPTION: If found, the **N**th character of the search string (**SEARCH$**) contained in the target string (**TARGET$**) is returned in **I%**. If not found, **I%**=0. For example, **.instr,”A”,B$,2** returns the position of the 2nd instance of the character “A” in the string **B$.**

Related Variables:
I% First character of **SEARCH$** found in **TARGET$**

See also: .PINSTR

.KEYMW KEY/MOUSE WAIT

Syntax: **.KEYMW**

DESCRIPTION: When called, the program stops and waits for either a mouse click or a key press. The mouse variables hold the mouse's current information. **I%** will contain the ASCII number of the key pressed (0 if none). For mouse clicks, **I%** is -1 for a left click and -2 for right.

Related Variables:
I% PET-ASCII value of key press *or* -1/-2 for left/right mouse click.

See also: .WKEY, ,KP

KEYPRESS KEY PRESS BUILT-IN

Syntax: **.KP,STRING$**

This routine quickly scans your string and checks if any of those keys are being pressed at the moment. If one is, that key's position within the string will be returned in **I%**.

Related Variables:
I% Key's position within the string

See also: .WKEY, .KEYMW

.LG LET GO

Syntax: **.LG**

DESCRIPTION: This DotCommand will wait until the user is not holding either of the mouse buttons (or their equivalents) down, even if it takes all day.

EXAMPLE

```
100 .do:.ma:.bg,x
110 x=(x+1)and15:.lg
120 .un L2%
```

See also: .CAGEM, .PUTM

.LNKSOB LINK SCREEN OBJECTS

Syntax: **.LNKSOB,LOC**

DESCRIPTION: After BLOADing a *Screen Object Collection*, **.LNKSOB** links the memory, defined in **LOC**, to the **DotBASIC** program. An alternative is to first use **.SETSOB,** then BLOAD the .SOC file.

See also: .BL, .CUTSOB, .DELSOB, .PSTSOB, .SETSOB

.MA MOUSE ASK BUILT-IN

Syntax: **.MA**

DESCRIPTION: Puts all the current conditions of the mouse in various variables.

Related Variables

PX%	Pixel-X Coordinate	**L2%**	New Left Click	**SC%**	Screen Code under Mouse
PY%	Pixel-Y Coordinate	**R1%**	Right Button State	**CC%**	Color Code under Mouse
CX%	Cell-X Coordinate	**R2%**	New Right Click	**PP%**	Screen Memory Position of Mouse
CY%	Cell-Y Coordinate	**RG%**	Region # Mouse is Over		
L1%	Left Button State	**CR%**	Region # being Clicked		

.MCMENU MULTI-COLUMN MENU

Syntax: **.MCMENU,NC,X,W,Y,I,U,H,HOT$**

H+128 Don't REVERSE/Un-REVERSE text.

DESCRIPTION: Displays a multi-column menu with **NC** number of columns (max 5). ***An X and W (width) parameter must be included for each column.*** The **Y** coordinate and **I** (# items) apply to all columns. **U** is the color of unhighlighted items in the menu. The highlight bar is colored **H**. If you don't want the text to REVERSE or un-REVERSE as the bar moves, add 128 to **H**.

The items are numbered in this order: down the first column, then the next, and so on. So, if you had 3 columns with 7 items in each column, the 2nd column would start with item number 8.

Items can be directly selected by pressing the appropriate *Hotkey* (**HOT$**). The highlight bar is moved to that item number, unless it doesn't exist. Pressing the *Global Escape* key (**MV+12**) ALWAYS returns a zero in **SL%.**

The selected item's number is returned in **SL%**.

Related Variables:

SL% # of the menu item selected.

See also: .MENU, .MENUA, .MENUB, .MSMENU, .SCMENU

.MENU MENU

Syntax: **.MENU,X,Y,W,I,U,HI,HK$**

HI+128 Don't REVERSE/Un-REVERSE text under Menu Bar
HI+64 Don't REVERSE/Un-REVERSE HotKeys

DESCRIPTION: Turns screen rows defined by you into menu lines. **U** is the color of unhighlighted items, and **H** is the color of the highlighted item. **HK$** allows us to define "hotkeys" for our menu. The number of the menu line chosen is returned in **SL%**

You can use the **CRSR/RETURN** keys instead of a mouse or joystick. In order to provide a more natural menu interface, the *Keyboard Enable* variable at **MV+20** is temporarily zeroed during menus. The **CRSR** keys are then read manually to move the highlight bar, one move per press, like we're all used to.

The **HOT$** string allows direct selection of menu items. For example, if you have hotkeys of "*loadst*r*", and the user presses "d", the fourth item is selected and the highlight bar is moved there. If there was no such item, **SL%** still returns a 4 but the highlight bar would not change.

The **H+64** feature of menus will only work if "*honor hotkey colors*" is enabled. A *Hotkey Color* is just ANY color within the text of a menu which is neither the highlight nor un-highlight color. That leaves you with 14 colors to make your menu's hotkeys stand out, and stay that way.

Refer to the ***Menu Madness*** section of the ***Tutorial*** for a through description of all the **DotBASIC Plus** menu features.

Related Variables:

SL% Selected item
MV+10 Menu Type

See also: .MCMENU, .MENUA, .MENUB, .MSMENU, .SCMENU

.MENUA AUTO-MENU A (SHADOW)

Syntax: **.MENUA,X,Y,U,H,HK$,ITEMS$**

DESCRIPTION: Automatically creates a menu (see **.MENU**) with shadow effect. Place the menu items in the **ITEMS$** string, with each item separated by an **F7**.

Related Variables:

SL% Selected item
MV+10 Menu Type

See also: .MCMENU, .MENU, .MENUB, .MSMENU, .SCMENU

.MENUB AUTO-MENU B

Syntax: **.MENUB,X,Y,U,H,HK$,ITEMS$**

DESCRIPTION: Automatically creates a menu (see **.MENU**). Place the menu items in the **ITEMS$** string, with each item separated by an **F7**.

Related Variables:

SL% Selected item
MV+10 Menu Type

See also: .MCMENU, .MENU, .MENUA, .MSMENU, .SCMENU

.MSG PRINT MESSAGE

Syntax: **.MSG,CO,STRING$**

DESCRIPTION: This command prints your string just like Region text. The screen line (0-24) the message will print at is defined in **MV+23**. You provide the color, and **MV+22** specifies the REVERSE state, centering, or the leading space. This command is useful for special prompts and messages. The plus "+" can be used to concatenate strings printed by this routine.

Related Variables:

MV+22 Region Text Color
+128 REVERSE printing
+ 64 leading space
+ 32 automatic center
MV+23 Region Text Row

See also: .DRTEXT, .EDRTEXT, .PRTEXT

.MSMENU MULTI-SELECT SCROLLING MENU

Syntax: **.MSMENU,X,Y,W,H,B,I,UN,HI,S,WB,LOC,T$,B$**

HI+128 Don't REVERSE/Un- REVERSE text under Menu Bar
HI+64 Don't REVERSE /Un- REVERSE HotKeys
W=255 File Requestor
X+128 Manual Icons
LOC=0 Do Not Rack

DESCRIPTION: Creates a scrolling menu that allows for multiple selections. The confusing letters are mostly colors and read like this:

Box
Icons
Un-highlight
Highlight
Selected
(selected) **W**ith bar
Location

It's always easy to spot the selected items, even when they are under the highlight bar. Which of the **U,H,S,W** items are REVERSEd is set from bits `3-0` to **MV+15**. By default, the highlight bar and all selected items are REVERSEd. Each REVERSE bit can be temporarily disobeyed by adding 128 to the **U,H,S,W** parameters.

Selecting items is as easy as hitting **RETURN** or clicking on them. The item is toggled and the mouse and highlight bar are moved down to the next item, scrolling when necessary - even when selecting with the mouse!

The additional keys **A, N**, and **T** function within multi-select menus to select **ALL, NONE**, and to **TOGGLE ALL** items (respectively). The mouse user cannot access these special features unless YOU enable manual icons!

Exiting a multi-select menu is the confusing part. If you press the **EXIT** key (**MV+16**), **SL%** will return the number of items selected. If **ESCAPE** is pressed, the menu is cancelled and zero is returned in **SL%**. The selected items still exist, and still can be indexed, but you are just told that there weren't any selected items.

Refer to the **.SEL** DotCommand to see how the selected menu items are accessed.

Related Variables
SL% Number of items selected
MV+10 Menu Type
MV+12 Global Escape
MV+16 Define "Quit" HotKey

See also: *.MCMENU, .MENU, .MENUA, .MENUB, .SCMENU,* .SEL

.P@ PRINT AT

Syntax: **.P@,X,Y,STRING$**

DESCRIPTION: Prints **STRING$** at location **X,Y**

See also: *.PC, .TEXT*

.PASTE PASTE

Syntax: **.PASTE,X,Y,W,H,LOC**

DESCRIPTION: Used with **.CUT**, this DotCommand requires that you specify the area to be filled. As long as you use the same **W**idth and **H**eight as the cut data, you can paste it wherever you want, oodles of times. However, no part of the area can be off-screen.

See also: *.CUT*

.PAUSE PAUSE

Syntax: **.PAUSE,JIFFIES**

DESCRIPTION: This routine waits for the specified number of jiffy interrupts to pass before returning control to you. The wait is (usually) measured in sixtieths of a second, and the value cannot exceed 255.

.PC PRINT CENTER

Syntax: **.PC,Y,STRING$**

DESCRIPTION: Prints **STRING$** centered on row **Y**.

See also: *.P@, .TEXT, .TEXTC*

.PINSTR PUT IN-STRING

Syntax: **.PINSTR, CHR$,TARGET$,POSITION**

DESCRIPTION: POKEs one byte of **CHR$** into **T$** at **POSITION. .PINSTR** changes the actual string itself. That is, if you have a short program:

```
10 t$=”text”
20 .instr,"x",t$,1:if i%=0 then 99
30 .pinstr,"s",t$,i%
99 .of:end
```

This program will search **T$** for any instances of “x” and then replace the “x” with an “s”. As expected, this changes the value of **T$** to “test”. What you may *not* expect is this – after RUNning the program, LIST it again:

```
10 t$=”test”
20 .instr,"x",t$,1:if i%=0 then 99
30 .pinstr,"s",t2$,i%
99 .of:end
```

.PINSTR changed the actual string in line `10`! If you want to preserve the original value of **T$**, concatenate **T$** with a “” (double quote), like this:

```
10 t$=”test”:t$=t$+””
20 .instr,"x",t2$,1:if i%=0 then 99
30 .pinstr,"s",t2$,i%:goto 20
99 .of:end
```

See also: .INSTR

.PPRNT PRINT RACKED DATA

Syntax: **.PPRNT**

DESCRIPTION: Prints previously Racked data to an attached printer (device 4), with 1 inch margins top, bottom, and left.

See also: .RK, .TEXRD

.PRFILE PRINT FILENAMES

Syntax: **.PRFILE,X,Y,INDEX#**

DESCRIPTION: If you've used Get Directory **(.DIR**) and have Racked up the resulting text, you can immediately print the filenames to the screen at **X,Y**.

See also: .DIR, .PRI, .RK

.PRI PRINT SELECTED ITEM

Syntax: **.PRI,X,Y,INDEX#**

DESCRIPTION: Indexes an item from racked data and prints it anywhere on the screen. The string is NOT returned in **W$** or **F$**.

See also: .RK, .PRFILE

.PRTEXT PRINT TEXT

Syntax: **.PRTEXT,INDEX**

DESCRIPTION: This prints Region text on the line specified in **MV+23**. It fills up the unused part of the line with spaces, so you don't have to worry about erasing the old text before printing over it. The string will be printed as specified in **MV+22**. Usually the "index" will be **RG%**, but it doesn't have to be.

```
100 .bl,0,”textfile”,d,40960
110 .edrtext,40960
200 .do:.ma
210 .prtext,rg%
220 .un cr
```

Related Variables:

MV+22 Region Text Color
- **+128** REVERSE Printing
- **+64** Leading Space
- **+32** Automatic Center

MV+23 Region Text Row

See also: .DRTEXT, .EDRTEXT, .MSG

.PSEL PRINT MULTI-SELECT MENU ITEM

Syntax: **.PSEL,X,Y,INDEX#**

Description: After exiting a multi-select menu, **SL%** holds the number of selections. To print out those selections:

```
100 if sl%>24thensl%=24
105 if sl%=0then:.of:stop
110 forx=1tosl%
120 .psel,0,x,x
130 next
```

See also: .DIR, .PRI, .SEL

.PSTSOB PASTE SCREEN OBJECT

Syntax: **.PSTSOB,INDEX#,X,Y**

DESCRIPTION: Pastes Screen Object to the screen. **INDEX#** is the number, in order of being cut, of the object, and **X** and **Y** are the coordinates for placing the upper left corner of the object to the screen. If any part of the Screen Object is off screen, or the index number is too high, the object will not be pasted onto the screen.

See also: .CUTSOB, .DELSOB, .LNKSOB, .SETSOB

.PUTM PUT MOUSE

Syntax: **.PUTM,X,Y**

DESCRIPTION: This DotCommand puts the mouse arrow anywhere on the screen.

See also: .CAGEM, .LG

.QR IRQ RESTORE BUILT-IN

Syntax: **.QR**

DESCRIPTION: Restores IRQ and the mouse pointer.

See also: .QS

.QS IRQ SUSPEND BUILT-IN

Syntax: **.QS**

DESCRIPTION: When **DB+** is started, an arrow is created (sprite at 44*256). You can turn off the arrow and mouse control at anytime with **.QS**, and turn it back on with **.QR**. When accessing the disk drive, it is often a good idea to execute a **.QS** beforehand. Note that this isn't necessary with any **DotBASIC** DotCommands, but if you are accessing the drive from **BASIC 2.0** (with the OPEN command, for example) you should suspend IRQ with **.QS** and then restore IRQ afterwards with **.QR**

See also: .QR

.RDMI RANDOM INDEX

Syntax: **.RDMI,ITEMS,BEGIN**

DESCRIPTION: Need to shuffle an index? Maybe you have a card game that needs 52 cards shuffled. Here is the fast way to do it in no time at all! **ITEMS** is the number of items to be shuffled (1-128), **BEGIN**ning with number (0-128).

NOTE: You must DIM an integer array at the top of your program (before any other DIM or use of any other array) with at least as many elements as items to be shuffled. This array will hold your shuffled index.

EXAMPLE

```
1 DIM R%(52)
100 .RDMI,52,1
110 FOR X = 1 TO 52
120 PRINT R%(X)
130 NEXT
```

R%(1) to **R%(52)** will hold the values 1-52 in random order.

.RESTR SCREEN RESTORE

Syntax: .**RESTR,PAGE**

DESCRIPTION: Restores the screen image stashed at given memory **PAGE**. Pages are an easy way to deal with memory. Each page is 256 bytes, so **PAGE***256 is the memory location. You can STASH and RESTORE to memory under ROM and I/O.

See also: .STASH

.RI RACK INDEX

Syntax: **.RI,INDEX#**

DESCRIPTION: Once you've racked up a ***Mr.Edstar*** file, you can *index* it. The indexed item is returned in **W$**. **F$** is also set by indexing, and will always return a null unless you happen to be looking at a directory, in which case it contains the entry's filename.

Related Variables

W$ Indexed Item
F$ Indexed Filename

See also: .PRI, .PRFILE, .RK, .RRK, .SAVSTR, .SETSTR

.RK RACK

Syntax: **.RK,LOC**

DESCRIPTION: This routine takes a ***Mr.Edstar*** file (terminated by a zero) that you have BLOADed into memory and "racks it up". By this, we mean that a table of pointers is created right after the zero at the end of the text, enabling you to use various DotCommands to grab or print individual lines of the file.

The file being racked can be located anywhere in memory, even under I/O. Racking needs 3 bytes per line at the end of the file for its pointers. The total number of items in the virtual array is returned in **N%**.

Related Variables
N% Number of Items

See also: .PRI, .PRFILE, .RI, .RRK, .SAVSTR, .SETSTR

.ROLOVR ROLL-OVER

Syntax: **.ROLOVR**

DESCRIPTION: Enables the automatic effects declared by **.SETROL** when the mouse rolls over a Region. See **.SETROL** for examples.

See also: .EVENT, .SETROL

.RRK RE-RACK

Syntax: **.RRK,LEN**

DESCRIPTION: When an area of memory is Racked by the **.RK** DotCommand, text is formatted to a length of 38 characters. This is a time-tested ***LOADSTAR*** standard that allows for very pretty text screens with room for a custom border around the text if so desired. If your project calls for a length other than 38 characters, use **.RRK** *after* you have Racked your data with**.RK**.

See also: .RK

.RU ARE YOU SURE?

Syntax: **.RU,BOX,REV,U,HI,UREV,STRING$**

DESCRIPTION: Puts a dialogue box in the center of the screen that asks, *"Are You Sure?"* The box is drawn with the color **BOX**. The unhighlighted YES/NO buttons are in the color **U,** highlighted YES/NO buttons are in the color **HI**. To REVERSE the dialogue box, set **REV** to 1.

If **REV** equals 1, the flag **UREV** can be set to one. This flag causes **.RU** to print the unhighlighted YES/NO option in REVERSE. To have the highlighted YES/NO option REVERSEd, add 128 to **HI.** If **REV** equals zero, setting **UREV** has no effect.

Color codes can be imbedded in **STRING$,** but centering will be off without some tweaking.

The user's response is placed in **YN%**.

Related Variables
YN% 0=NO
1=YES

See also: .YN

.SAVSTR SAVE STRING

Syntax: **.SAVSTR,STRING$**

DESCRIPTION: Stores BASIC strings into memory, (including under ROM or under I/O), which can later be Racked with **.RK**, indexed with **.RI**, saved to disk etc. **.SAVSTR** copies **STRING$** to the location defined by **.SETSTR**, followed by a zero byte. The next **.SAVSTR** begins at that zero byte.

FP holds the memory location of the zero byte. Thus, you can use these commands to create a text file! Assuming you have strings in **A$(n)** array:

```
100 .setstr,49152
110 for x = 1 to 10
120 .savstr,a$(x)
```

```
130 next
140 .bs,"t.text",d,49152,fp+1
```

Now you can load those strings into another array with:

```
200 .bl0,"t.text",d,160*256
210 .rk,160*256
220 dimb$(n%)
230 for x = 1 to n%
240 .ri,x
250 b$(x)=w$
260 next
```

Another good use is to "collect" filenames with a certain extension:

```
300 .dir,"$:*",d,160*256,240
310 .rk,160*256
320 .setstr,160*256
330 f=0:for X=1 to N%
340 .ri,X:if right$(f$,4)=".dbs" then:.savstr,f$:f=1
350 next
360 if f <> 1 then end
370 .rk,160*256
380 for x = 1 to N%:.ri,X:printw$:next
```

And yes! You can collect right on top of the directory data, because your string list will be smaller than the directory on every line.

Related Variables:

FP Memory location of last zero byte.

.SCMENU SCROLLING MENU

Syntax: **.SCMENU,X,Y,W,H,B,I,UN,HI,LOC,T$,B$**

HI+128	Don't REVERSE /Un- REVERSE text under Menu Bar
HI+64	Don't REVERSE /Un- REVERSE HotKeys
W=255	File Requestor
X+128	Manual Icons
LOC=0	Do Not Rack

DESCRIPTION: Creates a scrolling menu from the Racked data in memory at **LOC**. The extra parameters are:

Box
Icons
UN-highlight
Highlight
LOCation

T$** is printed at the top of the menu, and **B$ is printed at the bottom. If the Racked data is a disk directory, **T$** is defined by **DotBASIC** with the disk's header, while **B$** equals the "Blocks free" message.

Several things change when **X+128** is used.

1. The **BX,I,T$**, and **B$** parameters are ignored.
2. The scrolling menu is not drawn.
3. The mouse is not caged on the menu.
4. The **X,Y,W,H** parameters now represent the area for the *actual scrolling text* alone.

The following Regions can be defined by the programmer wishing to create their own menu buttons:

1. Home
2. Scroll up
3. Scroll down
4. Exit
5. Page up
6. Page down

For a very complete overview of this powerful DotCommand, refer to the ***Menu Madness*** section of the Tutorial.

See also: .DIR, .MCMENU, .MENU, .MENUA, .MENUB, .MSMENU, .PSEL, .SCNUME, .SEL

.SCNUME SCROLL NUMBER ENABLED

Syntax:**.SCNUME,CURX,CURY,TOTX,TOTY,SELX,SELY,REV**

DESCRIPTION: Like ***LOADSTAR's*** *Presenter*, you can make a message like "Line 172 of 308" that is updated each time the information changes. Multi-select menus can also show the total number of selected items.

To use **.SCNUME** simply define the X and Y cell coordinates of the info you want to display

CURX,CURY	Current Line Number
TOTX,TOTY	Total Lines
SELX,SELY	Number of Selected Items

Use a value of 255 in X or Y of any item you do NOT want to display.

Give **REV** a value of 1 to print the text REVERSEd. Otherwise make **REV** equal to zero.

Related Variables:
MV+15 Scroll Menu Type

See also: .MSMENU,.SCNUME

.SCPRNT SCRIPTOR PRINT INCLUDED WITH .SCRIPT

Syntax: **.SCPRNT,X,Y,STRING$**

DESCRIPTION: This DotCommand is Included with **.SCRIPT** and allows you to print **STRING$** to the **X** and **Y** coordinates on the 40 x 25 hi-res screen. You can't place the text at any pixel location. This routine prints as if the hi-res screen were a text screen.

See also: .SCRIPT

.SCRIPT SCRIPTOR

Syntax: **.SCRIPT,160,128,PAGE**

DESCRIPTION: In conjunction with **.SCPRNT**, this command allows the programmer to print text to the hi-res screen. **.SCRIPT** Initializes ***Scriptor*** with **.SCRIPT,160,128,FONT**, where **FONT** is the location of the text font. This is normally at page 8.

See also: .SCPRNT

.SEL INDEX SELECTED ITEMS

Syntax: **.SEL,NUMBER**

DESCRIPTION: After using a multi-select menu, **.SEL** is used to ask for each "N"th selected item.

EXAMPLE

```
10 for n=1 to sl%:.sel,n:print w$
40 next n
```

See also: .MSMENU, .PSEL

.SETROL SET ROLL-OVER INCLUDED WITH .ROLOVR

Syntax: **.SETROL,REGION#,U,HI**

REGION#=0	All Regions
HI=255	Disable Roll-Over for this Region

DESCRIPTION: Used with **.ROLOVR**, this DotCommand allows any defined Region to change color whenever the mouse arrow – ahem – rolls over them. **U** is the unhighlighted region color, and **HI** is the highlighted color.

See also: .EVENT, .ROLOVR

.SETSOB SET SCREEN OBJECT

Syntax: **.SETSOB,LOC**

DESCRIPTION: Defines location of the *Screen Object Collection* and zeroes the index. **LOC** is the beginning of the memory where the collection will be. This can be anywhere in memory, even under ROMs and I/O.

If a .SOC file is to be BLOADed, either do the BLOAD after **.SETSOB**, or use .LNKSOB.

See also: .CUTSOB, .DELSOB, .LNKSOB, PSTSOB

.SETSTR SET STRING LOCATION

Syntax: **.SETSTR,LOC**

DESCRIPTION: Sets the beginning of string memory. **.SAVSTR** can then be used to copy strings to that memory, followed by a zero byte.

See also: .PRI, .RI, .RK, .RRK, .SAVSTR

.SID SID PLAYER

Syntax: **.SID,LOC**
.SIDOFF

DESCRIPTION: Plays the ***SidSong*** located at **LOC** **.SID** uses pages 192-204, and the Top of **BASIC** is lowered from page 160 to 144 (unless **.BMP** is used). A good **LOC** for the MUS file(s) is at pages 144-154 (with **.BMP**) or 144-159 (without **.BMP**). **.SIDOFF** is automatically included with **.SID**.

See also: .SIDOFF

.SIDOFF SID OFF INCLUDED WITH .SID

Syntax: **.SIDOFF**

DESCRIPTION: Turns off music played by **.SID**

See also: .SID

.SPRITE SPRITE

Syntax: **.SPRITE,S#,0/1,I#,CO,X,Y**

I#=0	Sprite will not be changed
CO=16	Ignore color
X=0	Sprite will not be moved

DESCRIPTION: Puts any sprite anywhere on the screen. **S#** is the *Sprite Number* (0-7). **0/1** is *Off (0) or On (1)*. **I#** is the *Image Number*. (In **DB+**, Sprite Images can run from 185 to 223.) **CO** is obvious, as are **X** and **Y**. Remember, visible Sprite coordinates begin with **X**=24 and **Y**=50.

See also: .SPRFX, .SPRMV

.SPRFX SPRITE EFFECTS

Syntax: **.SPRFX,S#,XEX,YEX,PRI,MC**

DESCRIPTION: **.SPRFX** controls the various switches: **XEX** and **YEX** are *X-Expand* and *Y-Expand*. **PRI** is sprite *Priority*, and **MC** is *Multi-Color*. Using 0 disables an effect, 1 enables it. Putting 128 in any of these we leave the setting unchanged.

See also: .SPRITE, .SPRMV

.SPRMV SPRITE MOVE

Syntax; **.SPRMV,S#,X,Y,MODE**

DESCRIPTION: Allows you to link the positions of sprites to each other with given offsets. Sprite 0 cannot be linked, but each of the other sprites can be linked to the sprite just before it. For example, you have a moveable object that requires two sprites staying side by side. We will use sprites 0 and 1:

```
100 .SPRMV,1,24,0,1
110 .SPRMV,0,100,100,0
```

In line 100, we set **MODE** to 1 to link Sprite 1 to Sprite 0. The **X/Y** values are the pixel offsets — +24 and 0. The offsets can be negative numbers! Then, in line 110, we use **MODE** 0 to position Sprite 0 at 100/100. Sprite 1 will be placed at 124/100. The offsets continue until a non-linked sprite occurs (or Sprite 7). To unlink a sprite, use **MODE** 128.

See also: .SPRITE, .SPRFX

.STASH STASH SCREEN

Syntax: **.STASH,PAGE**

DESCRIPTION: Instantly stashes the whole screen to the given memory **PAGE**. We like using Page 208 or 216 —under I/O.

See also: .RESTR

.SWPMEM SWAP MEMORY

Syntax: **.SWPMEM,START,END+1,DESTINATION**

DESCRIPTION: Swap the area of memory with that at the destination, at a rate of 41 cycles per byte. Swapping to a location within the **START/END** range will have unwanted results!

See also: .CPYCHR, .CPYIO, .CPYMEM, .SWPMEM

.TEXRD TEXT READER

Syntax: **.TEXRD,LOC,BKGDCOL,TXTCOL,ICONCOL,NAME$**

DESCRIPTION: Racks the data in **LOC** and puts in on the screen. Includes option for 65-column (1 inch margin) print-out to printer 4.

BKGCOL is the background color, **TXTCOL** is the text color, **ICONCOL** is the color of the menu bar, title bar and up/down scroll icons, and **NAME$** is printed in the center of the title bar on row 0.
See also: .PPRINT

.TEXT TEXT BOX

Syntax: **.TEXT,X,Y,W,STRING$**

DESCRIPTION: This command prints the string at **X,Y** — and word wraps it to fit in **W** width.

See also: .BOX, .P@, .PC, .TEXTC

.TEXTC TEXT BOX, CENTERED

Syntax: **.TEXTC,Y,W,STRING**

Y+128 Center Vertically on Row Y

DESCRIPTION: Prints **STRING$** centered, beginning on row **Y**, word wrapped at width **W**. Add 128 to **Y** to center vertically on row **Y**.

See also: .BOX, .P@, .PC, .TEXT

.TX TEXT COLOR BUILT-IN

Syntax: **.TX,CO**

CO+128 REVERSE Text

DESCRIPTION: Changes the text color. Add 128 to **CO** for REVERSEd text.

See also: .BG, .BR

.UN UNTIL BUILT-IN

Syntax: **.UN,*Boolean expression***

DESCRIPTION: Used with **.DO**, this DotCommand causes the Do Loop to repeat until ***condition*** is true.

See also: .DO, WH

.WH WHILE BUILT-IN

Syntax: **.WH, *Boolean expression***

DESCRIPTION: Used with **.DO**, this DotCommand causes the Do Loop to repeat as long as ***condition*** remains true.

See also: .DO, .UN

.WKEY WAIT KEY

Syntax: **.WKEY**

DESCRIPTION: Halts the program until a key is pressed. The ASCII value is returned in **I%**.

Related Variables:
I% ASCII Value of Keypress

See also: .KEYMW

.YN YES/NO

Syntax: **.YN,X,Y,CO,HI,REV**

DESCRIPTION: Creates a 'Yes/No" dialog box anywhere on the screen defined by **X** and **Y**. **CO** is the box color and **HI** is the color of the YES/NO buttons when highlighted. To REVERSE the window, set **REV** to 1.

Related Variables:
YN% 0=NO
1=YES

See also: .RU

APPENDICES

IN THIS SECTION:

DotCommand Summary

Quick Reference Sheet

System Variables

Machine Language Protocol

Index

Final Thoughts

DOTBASIC COMMAND SUMMARY

Do-Loop	Page	Syntax
Do	51	.do *Start Do-Loop*
Loop Until	68	.un (boolean) *Loop Until = True*
Loop While	68	.wh (boolean) *Loop While = True*
Disk Access		
BLOAD	47	.bl,f$,d,loc
BLOAD with Zero	47	.bl0,f$,d,loc
BSAVE	48	.bs,f$,d,begin,end(+1)
Disk Command	50	.disk,com$,d
Get Directory	50	.dir,"$:*",d,loc,#filenames
Menu		
Auto Menu w/out Shadow	58	.menub,x,y,u,h,hk$,it$
Auto Menu w/Shadow	57	.menua,x,y,u,h,hk$,it$
Multi-Column Menu	56	.mcmenu,nc,x,w,y,h,uc,hc,hk$
Multi-Select Menu	58	.msmenu,x,y,w,h,b,i,u,hi,s,w,l,t$,b$
Normal Menu	57	.menu,x,y,w,i,u,h,hk$
Print Multi-Selection	61	.psel,x,y,index
Scroll Number Info	65	.scnume,cur,tot,sel
Scrolling Menu	64	.scmenu,x,y,w,h,b,i,u,hi,lc,t$,b$
Select Index	65	.sel,index
Screen Effects		
Are You Sure?	63	.ru,u,h,rev,yncolor,ynrev,string
Box -- Fancy	52	.fancy,x1,x2,y1,y2,s1,s2,c1,c2
Box – Regular	48	.box,x,y,w,h,sc,co
Color Background	47	.bg,co
Color Border	48	.br,co
Color Text	68	.tx,co
Event	52	.event ,keystring
Event Define Roll Text	51	.drtext,#,"literal string"
Event Def Roll Text Edstar	52	.edrtext,loc
Event Print Roll Text	60	.prtext,index
Event Region Affect	47	.areg,region#,sc,co
Event Region Define	51	.dreg,region#,x,y,w,h
Event Rollover	62	.rolovr
Event Set Rollover	65	.setrol,region,u,h
Print At	59	.p@,x,y,string
Print Center	59	.pc,y,string
Screen FTS Display	52	.fts,page
Screen Restore	62	.restr,page
Screen Stash	67	.stash,page
Swap Characters	49	.chrswp,seek,replace,co
Swap Colors	49	.colswp,seek,replace
Text Box Center Word Wrap	67	.textc,y,w,string
Text Box w/Word Wrap	67	.text,x,y,w,string
Text Cut	49	.cut,x,y,w,h,loc
Text Paste	59	.paste,x,y,w,h,loc
Yes/No	68	.yn,x,y,u,h,rev
Mouse		
Ask Mouse	56	.ma
Cage Mouse	49	.cagem.x,y,w,h
IRQ Restore	61	.qr
IRQ Suspend	61	.qs
Key/Mouse Wait	56	.keymw
Let Go	56	.lg
Put Mouse	61	.putm,x,y

DOTBASIC COMMAND SUMMARY, cont.

Memory Management	Page	Syntax
Copy Character	49	.cpychr,start,end+1,dest
Copy I/O Intact	49	.cpyio,start,end+1,dest
Copy Memory	49	.cpymem,start,end+1,dest
Swap Memory	67	.swpmem,start,end+1,dest
Virtual Array/Strings		
Instring Put	59	.pinstr,chr$,target$,position
Instring Search	56	.instr,search$,target$,beginnum
Rack Memory	62	.rk,loc
Rack Print Indexed Filename	60	.prfile,x,y,index
Rack Print Indexed Item	60	.pri,x,y,index
Rack Retrieve Index Item	62	.ri,index
Re-Rack	63	.rrk,len
Sort Directory	51	.dirsrt
Sort Racked Data Alpha	47	.alph,from
String Memory Put	63	.savstr,string
String Memory Set	66	.setstr,loc
Screen Objects		
Cut	50	.cutsob,x,y,w,h
Delete	50	.delsob
Location Link	56	.lnksob,loc
Location Set	66	.setsob,loc
Paste	61	.pstsob,index,x,y
Bitmap Graphics		
Graf Access Commands	53	.graf
Graf Clip	53	.gclip,x1,x2,y1,y2
Graf Fill	53	.gfill,x,y,pen
Graf Offset	54	.gr00,xoff,yoff
Graf Pen Color	54	.gpen,p0,p1,p2,p3
Graf Pixel Line	53	.gline,x,y,pen
Graf Pixel Plot	54	.gplot,x,y,pen
Graf Screen Mode	53	.gmode,mode
Screen Print	65	.scprnt,x,y,string
Screen Print Setup	65	.script,160,128,font
SHP Display	48	.bmpscr,1/0 on/off
SHP Load	48	.bmp,"file.shp",d,160,128,156
Input/Output		
Input	55	.inp,x,y,txcol,csrcol,len,def$
Input Enhanced	55	.inplus,x,y,t,c,len,st,out$,def$
Message Line	58	.msg,co,string
SID		
SID Player On	66	.sid,loc
SID Player Off	66	.sidoff
Sprite		
Font to Sprite	52	.f2spr,sceencode,sprimage#
Sprite Effects	67	.sprfx,s#,xex,yex,pri,mc
Sprite Move	67	.sprmv,s#,x,y,mode
Sprite On	66	.sprite,s#,0/1,i#,co,x,y
Miscellaneous		
Pause	59	.pause,jiffies
Random Index	61	.rdmi,begin (0-127),plus (0-128)
Wait for Keypress	68	.wkey

BOX / AFFECT REGION

.BOX,X,Y,W,H,SC,CO
.AREG,REG#,SC,CO

SC=255	PAINT
CO+16	BLOCK
CO+32	FRAME
CO+64	UN-REVERSE*
CO+128	REVERSE*
CO+192	FLIP*
CO=255	SHADE

Color RAM will not be affected unless you add an additional 16 to the values above.

**FRAME MV VALUES

MV+ 46	MV+ 42	MV+47
MV+44	MV+41	MV+ 45
MV+48	MV+43	MV+49

INPUT

.INP,X,Y,TXT,CSR,LEN,DEF$
.INPLUS,X,Y,T,C,L,ST,OUT$,DEF$

X+128	Reverse Input
Y+128	Numbers Only

TEXT BOX

.TEXT,X,Y,W,STRING$
.TEXTC,Y,W,STRING$

X+128	Center Horizontally
Y+128	Center Vertically on Y

MOUSE ASK VARIABLES

PX%	Pixel-X Coordinate
PY%	Pixel-Y Coordinate
CX%	Cell-X Coordinate
CY%	Cell-Y Coordinate
L1%	Left Button State
L2%	New Left Click
R1%	Right Button State
R2%	New Right Click
RG%	Region # Mouse is Over
CR%	Region # being Clicked
SC%	Screen Code under Mouse
CC%	Color Code under Mouse
PP%	Screen Memory Position

MENUS

.MENU,X,Y,W,I,U,HI,HOT$
.MCMENU,NC,X,W,Y,I,U,HI,HOT$
.MSMENU,X,Y,W,H,B,I,UN,HI,S,WB,LOC,T$,B$
.SCMENU,X,Y,W,H,B,I,UN,HI,LOC,T$,B$
.MENUA,X,Y,UN,HI,HOT$,ITEM$
.MENUB,X,Y,UN,HI,HOT$,ITEM$

EXAMPLE
.MENUA,0,0,8,9,"123C", "Item{F7}Item{F7}Item{F7}Close"

SL%	Selection Number
HI+128	Don't Reverse/Un-Reverse text under Menu Bar
HI+64	Don't Reverse/Un-Reverse HotKeys
W=255	File Requestor
X+128	Manual Icons
LOC=0	Do Not Rack
MV+12	Global Escape
MV+16	Define "Quit" HotKey

MV+10 (Regular Menus) / MV+11 (Multi-Column Menus)

+128	Automatic Caging of Mouse
+ 64	Automatic Point-to-First
+ 32	Must Select
+ 16	Escape Equal-to-Last **MV+11** = Click Any Active Region
+ 8	Honor Hotkey Colors
+ 4	Dual Response
+ 2	Un-highlight after Select
+ 1	Stray-to-Exit

Menu Regions

1. Home	6. Page down
2. Scroll up	7. Select all
3. Scroll down	8. Select none
4. Exit	9. Toggle all
5. Page up	10. Cancel

Menu Icons RAM Locations: 4734, 4739, 4742, 4746

MV VALUES

MV+0 Region Data Zone: LB	**MV+17** Twin Flag (S/T/M)
MV+1 Region Data Zone: HB	**MV+18** Keyboard Enable
MV+2 # of Active Regions	**MV+20** Region Text Pointers: LB
MV+3 Pointing Pixel-X	**MV+21** Region Text Pointers: HB
MV+4 Pointing Pixel-Y	**MV+22** Region Text Color
MV+5 Sprite Update	+128 **REVERSE** Printing
MV+6 Min. Joystick Speed	+ 64 Leading Space
MV+7 Max. Joystick Speed	+32 Automatic Center
MV+8 Joystick Acceleration	**MV+23** Region Text Row
MV+9 Joystick Deceleration	**MV+24** Index Pointer Zone: LB
MV+12 Global Escape	**MV+25** Index Pointer Zone: HB
MV+14 Right Keycode (F7)	**MV+26** Index Item Count: LB
MV+15 Scroll Menu Type	**MV+27** Index Item Count: HB
MV+16 Scroll Menu Exit	

MEMORY MAP

Page	Contents
128–131	BMP Color
132–133	SPRITES
134–145	*Grafstar*
146–152	MUS File
153–155	*Scriptor*
156–159	BMP Multi-Color Data
160–191	Bit-Map Data
192–204	*SIDPlayer*
205–207	Bit-Map Load/Unpack

*If you are not using **.BMP**, **.GRAF**, or **.SCRIPT**, then 144-159 are available for Music files, and 160-191 are available for data, such as directories and text.*

MISC

Calculate # Files for .DIR
INT((bufferspace-1)/32)

.TX,CO+128
Reverse Text

SYS DD+6
List DotCommands

DOTBASIC PLUS SYSTEM VARIABLES

MV CONTROLS / DEFAULTS

Variable	Description	Default
MV+0	Region Data Zone: LB	0
MV+1	Region Data Zone: HB	44
MV+2	Number of Active Regions	0
MV+3	Pointing Pixel-X	0
MV+4	Pointing Pixel-Y	0
MV+5	Sprite Update	1
MV+6	Minimum Joystick Speed	1
MV+7	Maximum Joystick Speed	8
MV+8	Joystick Acceleration	8
MV+9	Joystick Deceleration	80
MV+10	Menu Type	192

+128 automatic caging of mouse
+ 64 automatic point-to-first
+ 32 must select
+ 16 escape equal-to-last
+ 8 honor hotkey colors
+ 4 dual response
+ 2 un-highlight after select
+ 1 stray-to-exit

Variable	Description	Default
MV+11	Multi-Menu Type	192
MV+12	Global Escape	0
MV+14	Right Keycode (F7)	3
MV+15	Scroll Menu Type(n000uhsw)	135
MV+16	Scroll Menu Exit (Q)	81
MV+17	Twin Flag (sync/twin/mono)	128
MV+18	Keyboard Enable (rsl0000c)	193

+128 Return can click
+64 Space can click
+32 +128 Return can click
+64 Space can click
+32 C= Key can click.
+1 CRSR keys move arrow

Variable	Description	Default
MV+20	Region Text Pointers: LB	0
MV+21	Region Text Pointers: HB	4
MV+22	Region Text Color	1

+128 REVERSE printing
+ 64 leading space
+ 32 automatic center

Variable	Description	Default
MV+23	Region Text Row	24
MV+24	Index Pointer Zone: LB	0
MV+25	Index Pointer Zone: HB	0
MV+26	Index Item Count: LB	0
MV+27	Index Item Count: HB	0
MV+41	Frame Center Area	32
MV+42	Top Edge	64
MV+43	Bottom Edge	64
MV+44	Left Edge	93
MV+45	Right Edge	93
MV+46	Top-Left Corner	112
MV+47	Top-Right Corner	110
MV+48	Bottom-Left Corner	109
MV+49	Bottom-Right Corner	125

MOUSE VARIABLES

Variable	Description
PX%	Pixel-X Coordinate
PY%	Pixel-Y Coordinate
CX%	Cell-X Coordinate
CY%	Cell-Y Coordinate
L1%	Left Button State
L2%	New Left Click
R1%	Right Button State
R2%	New Right Click
RG%	Region # Mouse is Over
CR%	Region # being Clicked
SC%	Screen Code under Mouse
CC%	Color Code under Mouse
PP%	Screen Memory Position of Mouse

DOTBASIC MACHINE LANGUAGE PROTOCOL

ML Coders *UNITE!* You have nothing to lose but your Chains!

OK, maybe "chains" is too strong a term. But if you code in 65xx ML, you know how hard it is to even think in terms of relocatable code. Our ML likes absolute addresses. Which is not a particularly bad thing. I mean, we only have 65536 bytes to worry about!

DotBASIC Plus is designed for those who have not become addicted to individual bits and bytes, but want plenty of power over the machine. And since **DB+** is Adaptable to the needs of the programmer, we can provide infinite DotCommands to do just that.

But we have to follow a certain "protocol", a few rules that make **DB+** ML code "modulettes" work with each other and the system. So this article is about how to write a new DB+ Command.

CHOOSING A NAME

Obviously, the command name has to be unique, a legal filename, no more than 9 characters, and not include any **BASIC 2.0** Keyword. *(You won't believe my own frustration with command names like **.PRINTSTR**! The PRINT is tokenized by **BASIC 2.0**, and is not recognized by **DotBASIC**. Moan! —Dave)*

The filename for the code must be

DCM.*command*.ML

where ***command*** is then name of your new DotCommand.

THE LAYOUT

When a DotCommand is called from the **DB+** program, the Carry is Cleared. When the same command is called from another ML command, the Carry is Set. This allows us to sort out the two situations in just four bytes of code:

```
0000 BCC BAS
0002 BCS ML
```

Often, the difference between the two is that the **BAS** section will collect parameters from BASIC and put the values in various places. The **ML** portion expects the data to be in such places.

```
BAS
**** JSR GETINT
**** STA 900
**** JSR GETINT
**** STA 901

ML
**** LDA 900
**** LDX 901
        etc.
```

But getting back to the header

```
0000 BCC BAS
0002 BCS ML
0004 BYT 255
0005 BYT ",S,Y,N,T,A,X",0

ML
0006 RTS

BAS
```

```
0007 ....
```

Byte 4 of the code is the beginning of the **ML**'s "include" section, which ends with a 255 byte. We will look at this in a moment.

Next comes the "self-documenting" feature. In the above example, the text is added to the command name (when the **DB+** programmer types **SYS DD+6**) as

```
COMMAND,S,Y,N,T,A,X
```

Your code at **BAS** will need to get the parameters into the code. We have some jumps to do this for you:

DD = 14336

GETINT Gets Integer
Returns: LO>.A HI>.X
(.Y not affected)
JSR DD+15

GETFP Gets Floating Point
Returns Floating Point in ACC#1
JSR DD+12

GETSTR Gets String
Returns: LEN>.A LO>.X HI>.Y
JSR DD+9

GETSIN Gets Signed Integer
Returns: LO>.A HI>.X
Allows negative values.
JSR DD+58

DOING DATA

You can put transitory data anywhere you think will be safe. I use the cassette buffer extensively for put and take situations. Obviously, this area is no good for values I want to keep from command call to command call. For this, you can set aside space in your code itself, using the following method:

```
DD=14336
SETLOC = DD + 27

TOP
   BCC BAS
   BCS ML
   BYT 255
   BYT",x,y",0
ML
   RTS
DATA
   BYT 0,0,0,0
BAS
   LDA#DATA-TOP
   LDX#164
   JSR SETLOC

   JSR GETSIN
   LDY#0
   CLC
   ADC(164),Y
   STA(164),Y
   INY
   TXA
   ADC(164),Y
   STA(164),Y
   JSR GETSIN
```

```
        LDY#2
        ADC(164),Y
        STA(164),Y
        INY
        TXA
        ADC(164),Y
        STA(164),Y
```

Here we set aside four bytes at **DATA**. To "find" where these bytes are, we first put the relative location of **DATA** (**DATA-TOP**) in .A, and a zero page address in .X. Then we use **SETLOC**, which finds the current address of the code and adds the distance to **DATA**. The result is put in the zero page address given in .X and the next — in this case 164/165, ready for Indirect Y addressing!

And that is just what we do with each **GETSIN** — adding the value to each byte in **DATA**. Later, we can use the same data to position a sprite or the cursor.

But, you are saying, *you have several commands that must access the same safe data. That certainly won't work with **SETLOC***. And you are right! So we have Data Blocks. These are small PRG files with enough space for the needed shared data. A Data Block filename uses a "#" to signify it is data and not a command. So use your assembler to create the Data Block file, with a name such as:

```
DCM#MYDATA.ML
```

In your code, you will need to include the Data Block.

```
DD = 14336
SETDAT = DD + 30

   BCC BAS
   BCS ML
   BYT 1,"#MYDATA",0,255
   BYT ",Q",0
ML
   RTS

BAS
   LDA#1  ; Index > .A
   LDX#164; ZP > .X
   JSR SETDAT
```

The Data Block name **(#MYPROG**) is preceded by a number byte which serves as an index. The name is followed by a 0 byte and the "include" area is terminated with a 255 byte.

To set the Data Block, the index number is put in .A and the zero page address is put in .X. **JSR SETDAT** then put the address of the Data Block in the two zero page bytes, ready for an Indirect Y call.

SUBROUTINES

If you absolutely need a subroutine in your code, you can use the **SETLOC** to put the subroutines address in two zero page bytes. Then POKE a 76 in the byte before the **ZP** address bytes. Simply **JSR** to that **ZP** location.

```
TOP
   BCC BAS
   BCS ML
   BYT255
   BYT0
ML
   RTS
ROUTINE
   LDA#0
   SEC
LOOP
   ROL
   DEX
   BPL LOOP
   RTS
```

```
BAS
   LDA#ROUTINE-TOP
   LDX#165
   JSR SETLOC
   LDA#76
   STA164

   JSR GETINT
   AND#7
   TAX
   JSR 164
```

This code uses a subroutine to set a given bit (bit number in .X). At **BAS**, we put the relative address of **ROUTINE** in .A and 165 in .X. The location of **ROUTINE** is put in 165/166 by **SETLOC**. Then we put 76 in 164 (**JMP** instruction), and we are ready to roll!

COMMAND CALLING COMMAND

But now for the niftiest trick of **DB**+ ML protocol. One command can call the ML portion of another command!

Let's first write **DCM.SETBIT.ML**

```
   BCC BAS
   BCS ML
   BYT255
   BYT":ML BIT>900",0
   BAS
   RTS
ML
   LDA#0
   SEC
   LDX 900
LOOP
   ROL
   DEX
   BPL LOOP
   RTS
```

Simple enough. Now we will write **DCM.SPR1.BC**, which will turn on a given sprite.

```
DD = 14336
SETML = DD + 21
DOML = DD + 24

    BCC BAS
    BCS ML
    BYT 1,"SETBIT",0,255
    BYT ",BIT#",0
ML
    RTS

BAS
    LDA#1
    JSR SETML
    JSR GETINT
    STA 900
    JSR DOML
    ORA 53269
    STA 53269
    RTS
```

Note that in the include section, **SETBIT** does NOT have the preceding dot. I did this just to confuse myself! Again, the index number is put in .A then **SETML** is called. This sets up the jump to the code of **SETBIT**. Then any data can be set up for the routine (putting the bit number in 900) and the routine called with **DOML**.

SUMMARY

That's all there is to it. Do not jump directly to any location in your code. Do not load or store a register directly from or to any location in the body of your code. But you do have everything you need to put relative locations in zero page or find and use other command codes. Here is a schematic of the first bytes of your code:

```
BCC BAS
BCS ML
BYT<index byte>
BYT<"com-name" or "#data-name">
BYT<0>
BYT<another index byte>
BYT<"com-name" or "#data-name">
BYT<0>
BYT 255    ; end include section

BYT<"syntax info">
BYT 0

Open space

BAS
   ...
   RTS

ML
   ...
   RTS
```

(I am looking forward to any and all DotCommands you might think of. I believe anything we can do in ML can become available to ***DB+*** *users! – Dave)*

INDEX

The DotCommands

.ALPH, *45*
.AREG, *14*, *45*
.BG, *8*, *34*, *45*, *52*
.BL, *23*, *33*, *39*, *41*, *45*, *50*
.BL0, *23*, *39*, *45*, *61*
.BMP, *33*, *34*, *46*, *51*, *52*, *64*
.BMPSCR, *33*, *34*, *46*, *52*
.BOX, *11*, *12*, *19*, *46*
.BR, *8*, *46*
.BS, *46*, *61*
.CAGEM, *47*
.CHRSWP, *47*
.COLSWP, *47*
.CPYCHR, *47*
.CPYIO, *47*
.CPYMEM, *47*
.CUT, *40*, *47*
.CUTSOB, *40*, *48*
.DELSOB, *41*, *48*
.DIR, *23*, *48*, *62*
.DIRSRT, *49*
.DISK, *48*
.DO, *9*, *49*
.DREG, *13*, *49*
.DRTEXT, *49*
.EDRTEXT, *50*
.EVENT, *50*
.F2SPR, *50*
.FANCY, *50*
.FTS, *41*, *50*
.GCLIP, *34*, *35*, *51*
.GFILL, *34*, *35*, *51*, *52*
.GLINE, *34*, *51*, *52*
.GMODE, *34*, *51*, *52*
.GOFFSET, *34*
.GP, *34*, *35*, *52*
.GPEN, *34*, *52*
.GPLOT, *34*, *51*, *52*
.GR00, *35*, *52*
.GRAF, *33*, *34*, *51*, *52*
.I2FP, *23*, 45, *52*
.INP, *53*
.INPLUS, *53*
.INSTR, *54*
.KEYMW, *12*, *13*, *54*
.KP, *16*, *54*
.LG, *54*
.LNKSOB, *40*, *54*
.MA, *9*, *54*
.MCMENU, *22*, *55*
.MENU, *19*, *55*
.MENUA, *55*
.MENUB, *56*
.MSG, *56*
.MSMENU, *28*, *56*
.OF, *8*
.P@, *14*, *57*
.PASTE, *40*, *57*
.PAUSE, *57*
.PC, *14*, *57*
.PINSTR, *57*
.PPRNT, *58*
.PRFILE, *58*
.PRI, *39*, *58*
.PRTEXT, *58*
.PSEL, *59*
.PSTSOB, *41*, *59*
.PUTM, 14, 15, *59*
.QR, *59*
.QS, *10*, *59*
.RDMI, *60*
.RESTR, *19*, *41*, *60*
.RI, *39*, *60*, *61*
.RK, *39*, *60*, *61*
.ROLOVR, *61*
.RRK, *61*
.RU, *61*
.SAVSTR, *61*, *62*, *64*
.SCMENU, *23*, *62*
.SCNUME, *63*
.SCPRNT, *36*, *63*
.SCRIPT, *36*, *63*
.SEL, *29*, *63*
.SETROL, 50, *63*
.SETSOB, *40*, *64*
.SETSTR, *61*, *62*, *64*
.SID, *33*, *64*
.SIDOFF, *33*, *64*
.SPRFX, *65*
.SPRITE, *64*
.SPRMV, *65*
.STASH, *19*, *65*
.SWPMEM, *42*, *65*
.TEXRD, *65*
.TEXT, *12*, *13*, *65*
.TEXTC, *66*
.TX, *8*, *25*, *66*
.UN, *9*, *66*
.WH, *10*, *66*
.WKEY, *66*
.YN, 66

A

Alphabetize, *45*

B

B.DOTBASIC, 7
B.Files, *7*
Bitmap Graphics, 33
BLOAD, *22*, *23*, *33*, *39*, *41*, *46*, *50*
Block, *45*, *46*
Boxes, *12*, *45*

C

Cage, *27*
Colors
  Screen, Text, Border, 8

D

DBDesign, *36*, *40*, *41*, *50*
DEV, *11*
Disk Blocks, *6*
Disk Menu
  DotMENU Project, *24*
DML Files, *7*, *11*
Do-Loop, *9*, *12*
DotBASIC
  Adding new DotCommands, *10*
  Creating new DotCommands, *71*
  Editing, *9*
  Features, *4*
  History, *2*
  Quiting, *8*
  Registering, 42
  Starting a new project, *7*

E

EDSTAR, *23*, 39, *48*, *50*
Error Messages
  B.DEV troubleshooting, *11*
  DOTCOM NOT FOUND ERROR, *11*

F

Filenames, collecting, *62*

Fill, *36*
Fonts
  Creating and replacing, *41*, *42*
Frame, *45*, *46*

G

Get Directory, *23*
Global Escape, *22*, *24*, *27*, *55*, *70*
GOTO60000, *9*
Grafstar, *33*

H

Hotkeys, *26*, *28*, *55*

I

***Including* DotBASIC Commands**, *10*, *11*, *46*
INDEXing Data, *28*, *39*
Input, *53*
  Keypresses, *12*, *16*, *54*

J

Joysticks, *9*, *30*

K

Key/Mouse Wait, *12*
Keycodes, *26*
Keypresses, detecting, *12*, *54*

L

Loading Files. See BLOAD
LOADSTAR, 2

M

Machine Language, *71*
Manual Icons. *See* Menus, Manual Icons
Memory Management, *6*
Menus
  File Requestors, *23*
  Manual Icons, *29*
  Multi-Column, *22*
  Multi-Select Scroll Menu, *28*
  Scrolling Menu, *23*, *30*
  Scrolling Menus, *22*
Mouse
  Button Regions, *13*
  Buttons, *26*
  Reading, *9*, *10*, *54*
  Variables (MV), *26*
Mouse Ask, *9*, *54*
MOUSE2.1 7K 1000, *7*
Mr.Edstar, *23*, *39*, *60*
Mr.Mouse, *28*
Music, *33*, *See* **.SID**
MV Variable, *26*
  Complete List, 70
MV+10, *27*
MV+11, *27*
MV+12, *27*

P

Page in RAM, *4*, *6*
Paint, *45*, *46*
PRINTing Text, *12*, *39*, *56*
PRINTint to Bitmap, *36*

R

Racking Data, *39*
Region Text, *49*
Regions, *13*, *27*, *28*, *29*, *30*, *45*, *49*, *50*, *56*, *58*
  Buttons, clickable, *13*
  Maximum Number, *13*
  Scrolling Menu Regions, *29*
Roll-Overs, *15*, *50*, *63*

S

Scratch and Save Routine, *9*
Screen Design, *41*, *50*
Screen Objects, *40*
Scriptor, *36*
Scroll Number, *63*
Scrolling Menus, *22*
Shading Effect, *15*, *21*, *46*
SHP Graphic File, *33*
SIDPlayer, *33*
Sprites, *50*, *64*

T

Text Box, creating, *12*
Text Boxes, *12*
Tool-Box-Stash, *41*

V

Variables
  B$, 23, *24*, *62*
  db, *3*
  dw, *3*
  F$, *39*, *60*
  Mouse Ask, *54*
  N%, *39*
  SL%, *20*, *24*
  T$, 23, *24*, *62*
  Typical Parameters, *3*
  W$, *29*, *39*, *60*
Virtual Arrays, *39*

W

Windows, *12*
Word-Wrapping Text, *12*, *65*
Work Disk, *4*, *7*, *33*

FINAL THOUGHTS: WHERE TO GO FROM HERE

Editing and writing new material for this reference guide has been a wonderful way for me to learn the ins-and-outs of **DotBASIC Plus**. If you want to really dig in deep, I highly recommend you spend the next six months or so re-writing and editing this book.

Barring that, here are a few things this novice programmer has learned on his journey:

- ***I love DBDESIGN***. With this astounding tool you can easily create some pretty amazing looking screens before typing a single line of code. There are several FTS files on the **DotBASIC** disks. If you are a **LOADSTAR** reader you will find all sorts of interesting FTS screens, fonts, and SHP files in practically every recent issue. I have learned quite a few interesting tricks just from taking a look at the way Dave creates the FTS screens he uses with **LOADSTAR**. Every program I write has a nice FTS screen – and there's no reason every program *you* write can't also have a great looking interface.
- ***To make defining your Regions simpler, use DBDESIGN.*** Draw boxes (Edit/Box) around your Regions, and you will see the *area* parameters in the top right of the screen. Just write these down for each region and you can create your Regions with no guesswork!
- ***After BLOADing your programs FTS file, use .STASH to tuck your screen away in memory.*** Now you can use those 16 pages of RAM occupied by the FTS file for other things.
- ***DBDESIGN can also create TBS files.*** These are like FTS files, minus the font information. Thus, they are only 8 pages long rather than 16. To use a TBS file in your **DB+** program, BLOAD the TBS into RAM and then **.restr,PAGE** where **PAGE** is the location where you BLOADed the TBS.
- ***Screen Objects*** are useful for many applications. Everything from game characters to drop down menus can be built from Screen Objects and then easily moved around the screen. Again, **DBDESIGN** is *the* tool for creating Screen Objects. Design the Objects, then copy them to your Screen Object Collection (Edit/Box/Copy/Store SOB). When you have your collection designed, save it as a *Screen Object Collection*.
- ***To replace the font in your DB+ program***, BLOAD the new font then swap it with pages 8 – 16. If your font is BLOADed to page 224, for example, just **.swpmem,8*256,16*256,224*256.** Do the same swap again to switch back.
- ***LINK and PACK your DotBASIC projects.*** These two utilities, both on the **DB+ Utility Disk** are quite useful. LINKER will take the separate files that make up your **DB+** project and combine them into a single file (don't include the B.FILE), then PACK will compress that single file to the smallest possible size.
- ***When using .SID, remember that you can't put anything in pages 192 – 204*** (49152 – 52224), as **.SID** makes use of this area. Also, by issuing PEEK(49152) and checking for a non-zero value, you can determine if the SID is still playing.
- ***Use the DO-LOOP!*** Anytime you are about to write a FOR-NEXT loop, stop and ask yourself if it couldn't be done more efficiently with a DO-LOOP. You'll find that DO-LOOPs execute much faster than FOR-NEXT loops.
- ***Read this book!*** Working through the tutorials in the beginning will get you started, and the **DotBIBLE** and **Quick Reference Sheet** should be a constant companion when coding.
- ***Subscribe to LOADSTAR*****.** You'll find several new DotBASIC programs in each issue, along with documentation.
- ***Visit the DotFORUM!*** When you purchased DB+ an account was created for you on the DotBASIC Forum, online at ***http://8bitcentral.com/dotbasic***. You'll find further tutorials (including videos), places for you to ask questions and get answers, and receive DB+ updates and programs.

Happy Programming!

ARR