GETTING STARTED

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
```

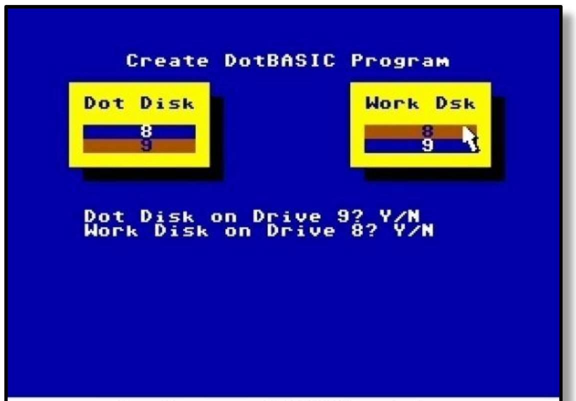
And here is the cool part. On the disk directory, we can see how many Disk Blocks a file uses. A Disk Block is 254 bytes long, so we can use this number to figure out where to put another file. For instance, if "t.text" is 12 Blocks in size, we know we can BLOAD "t.text2" to page 160+12, or

```
.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.



```
Dot Disk on Drive 9? Y/N
Work Disk on Drive 8? Y/N
```

Creating a new DotBASIC project

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*

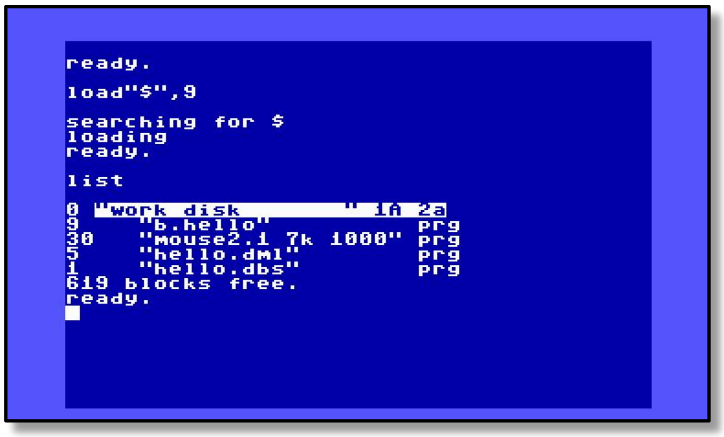
And

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.

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:



```
ready.
load"$",9

searching for $
loading
ready.

list

0 "work disk        " 1a 2a
9    "b.hello"          prg
30   "mouse2.1 7k 1000" prg
5    "hello.dml"        prg
1    "hello.dbs"        prg
619 blocks free.
ready.
```

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.

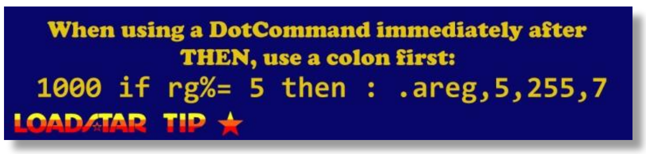
HELLO.DML is the ML program that interprets commands from **DB+** and makes them work. One of the neat things about **DotBASIC Plus** is that the .DML program only contains the DotCommands your program needs—you won’t waste valuable RAM on DotCommands you aren’t using *(see page 10 for more on this).*

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
```

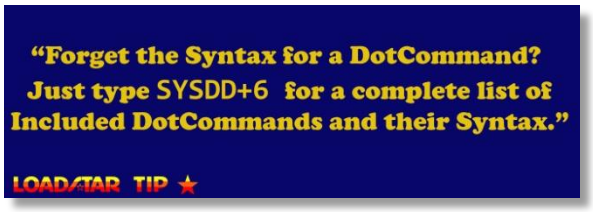
Change line 102 to

```
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!


"Forget the Syntax for a DotCommand? Just type SYSDD+6 for a complete list of Included DotCommands and their Syntax."

LOADSTAR TIP ★

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.

Don't use spaces when 'Including' DotCommands.

```
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
```

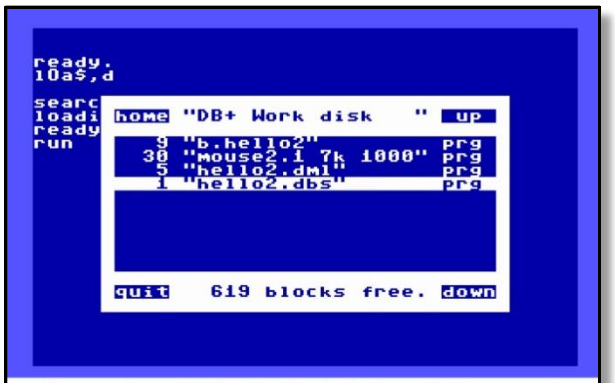
Notice lines 10 and 20. To "include" new DotCommands, all we do is enter them between these two lines, in the form of **REM** statements.

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.



```
ready.
10a$,d

searc
loadi  home  "DB+ Work disk      "  up
ready
run          9  "b.hello2"          prg
            30  "mouse2.1 7k 1000"  prg
             5  "hello2.dml"        prg
             1  "hello2.dbs"        prg

       quit    619 blocks free.   down
```

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!

"If DEV stops, it means you 'Included" a non-existant DotCommand, forgot the REM, or put a space before the dots."

LOADSTAR TIP ★

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.*