The 7800basic Developers Guide

This document is meant as an introduction to the 7800basic language command structure and syntax. Its not meant as a general programming primer.

Introduction

7800basic is a programming language that can be used to create games for the Atari 7800 game console. It differs from the majority of BASIC languages in that it compiles your source code into fast 6502 machine language code.

7800basic is based on the Atari 2600 programming language batari Basic, so if you're familiar with bB, you'll be right at home with 7800basic.

7800basic is designed to put as much control as you want in your hands, so if you're familiar with 6502 assembly code you can easily customize the modular framework, or mix your high level BASIC source code with your own low level assembly code.

If you don't want to learn assembly language, don't worry, you don't need to know assembly language to produce games. Everything you need is accessible using pure BASIC.

7800 Hardware Overview

Although 7800basic handles many of the hardware details for you, it will help to have a general understanding of the hardware you'll be developing for.

Controls

The 7800 comes with 2 dual-button joysticks, though the console is often used with Atari single button joysticks as well. The console can also be used with other legacy Atari controls, such as paddle

controls, driving controls, and keyboard controls.

The 7800 console itself has a number of switches on the console: Power, Pause, Reset, Player 1 Difficulty A/B, and Player 2 Difficulty A/B.

The Power button is a hardware switch that can't be detected or otherwise interacted with through software.

The Pause button is a momentary switch. If you wish to support in-game pausing, your game code needs to check this switch every frame. If it's pressed and released, your game code should stop the action. If pressed and released again, the game code should resume the action.

The Difficulty Switches can either be in A or B position. If you wish to support these switches in your game, you should check these switches during gameplay, and adjust difficulty as selected. (A=pro, B=amateur)

Graphics

The graphics chip in the 7800 is called MARIA. MARIA was designed to with an architecture similar to many arcade games of its era, and is unlike any other console graphics chip.

Sprites and Characters

MARIA takes a series of instructions to draw sprites and characters, and runs though them to create the screen bitmap display. In a sense, MARIA is hardware assistance for soft-sprites.

If MARIA runs out of time to create the screen bitmap, sprites and characters it didn't reach aren't displayed. The number of graphical objects you can successfully put on the screen depends on the width and color depth of the graphics you specify.

Another consequence of the screen being a bitmap is that MARIA provides no hardware collision detection. Your game code will need to do its own check of overlapping rectangles to see if sprites have collided.

Palettes

MARIA is equiped with 8 adjustable 3-color palettes. A sprite or character can be drawn using any one of these palettes, allowing for fairly colorful screens.

Graphics Modes

MARIA is capable of 3 graphic display modes, 160A, 320A, and 320B. Each of these graphic modes is capable of display sprites or characters in its regular format, or an alternate format.

Here's a quick summary table with the mode details...

Mode	Resolution	Number of Colors (in addition to transparent)	Width of One Character	Alt. Format	Alt. Number of Colors (in addition to transparent)	Alt. Width of One Character	Restrictions
160A	160x192	3 colors	4 pixels	160B	12 colors	2 pixels	None
320A	320x192	1 colors	8 pixels	320C	3 colors	4 pixels	In 320C mode, even pixel pairs use the first two colors in the palette, and odd pixel pairs use the last two colors in the palette.
320B	320x192	3 colors	4 pixels	320D	3 colors	8 pixels	320B mode requires pairs of pixels to use the same palette. 320D requires pixels in odd columns to use the first two palette colors, and pixels in even columns to use the last two palette colors.

As you can see from the table, the 160A/B mode has no restrictions, and is generally the most popular. 320A and 320B are also particularly useful and allow for higher resolution images.

The alternate formats 320C and 320D put odd limits where certain colors can be used, and as a consequence they're used less often than the other modes.

Zones

MARIA, splits the screen up into a series of horizontal bands called zones. A 7800 game typically spends a good deal of its time setting up memory structures for MARIA, called display lists. Display lists tell MARIA which characters and sprites it should be displaying in any one zone.

7800basic will deal with zones and create and manipulate display lists on your behalf, so there won't be further mention of them in this guide.

Sound

The 7800 uses the TIA chip for audio, just as the Atari 2600 does. This chip is capable of unique sounds, though it has coarse frequency control, which makes it difficult to play musical notes that sound in tune.

Its possible for the 7800 to play sound through an in-cartridge sound chip. The POKEY sound chip from Atari is the typical choice for this, and was used in some commercial games, such as Ballblazer and Commando.

7800basic Language

Formatting and Layout Features

indenting and white space

Each command in 7800basic should be indented, that is, there should be one or more spaces or tabs between the command and the left-margin. If you forget to do this, your basic program will compile incorrectly and fail miserably.

In fact, the only things that aren't indented in 7800basic are labels (we'll talk about them in a second) and multi-line command “end” markers. (we'll get into those with specific command documentation)

numeric representation

7800basic programs can use numbers in decimal, hexadecimal, and binary formats. Numbers with a $ prefix are assumed to be in hexadecimal format, numbers with a % prefix are assumed to be in binary format, and numbers without a prefix are assumed to be in decimal format.

Here's an example of decimal, hexadecimal, and binary numbers.

```
13
$2A
%00110011
```

line numbers and labels

7800basic doesn't require the old-fashioned BASIC style of numbering each line – in fact we don't recommend it - but it does support line numbers if you prefer to use them.

Instead of using line numbers, your 7800basic program can use labels for important sections of code. A label is a name you give a section of code, and is easily identified because it has no indentation.

If you wish to jump to a certain section of code, you can use goto with the label name. This is much easier to read and understand than using line numbers. Which of these examples do you prefer to read, amongst hundreds of similar commands?

```
goto BlowUpEnemy

goto 12428
```

We'll touch more on goto and related commands when we discuss program flow control.

combining multiple commands

In 7800basic you may use the “:” character to chain several commands together on a single line. This is most useful in if...then commands, when you'd like to do many things conditionally.

```
If PlayerY>150 then PlayerDying=1:PlayerDeathSound=1:goto GameOver
```

Variables

regular variables

Regular variables in 7800basic are byte sized, and limited to values from 0-255. If math on a regular variable causes it to exceed 255, it will overflow and become a small number near 0. Similarly, if math on a regular variable causes it to decrease below 0, it will underflow and become a large number near 255.

dimensioning variable names

You have 126 pre-named variables in 7800basic, ready and available exclusively to your game code. These have been given the names a-z and var0-var99, but you're not stuck with those names. Defining a new name is as simple as using the dim command.

```
dim MyPlayerDied=q
```

If 126 variables isn't enough memory for your game, there's also a 1.5K block of unnamed ram locations that's ready for use in your game. This block ranges from memory address $2200 to $27FF, and you can assign any bit of it a name using the dim command.

```
dim UfoState=$2200
```

So every time “UfoState” is used in code, the 7800 will really operate on memory location $2200.

assignment

Assigning a value to a variable is just a matter of using the form: “myvariable=myvalue” where myvalue can be a number, another variable, or a complex expression.

```
UfoState=0
Laserx=PlayerX
frame=(frame+1)&63
```

Expressions can contain operators for addition, subtraction division, multiplication, and/or bitwise operators.

The order of operations for expressions is:

```
(  )        brackets
*  /  //    multiplication, division, and modulus
+  -        addition, and subtraction
&  |  ^     bitwise AND, bitwise OR, and bitwise XOR
```

other variable types

variable arrays

Regular variables can be accessed as arrays in 7800basic. Doing so will access memory locations next to the variable in question.

The following sets variables “a”, “b”, and “c” to 0.

```
a[0]=0
a[1]=0
a[2]=0
```

Using array notation will allow you to loop through elements without a lot of duplicated code, providing you've used dim to place the elements side by side in memory.

Note: many advanced statements in 7800basic don’t work directly with arrays. To use arrays with those statements, you will need to assign any array lookups to a variable, and use the variable with those statements.

bitwise variables

7800basic also has the ability to work directly with bits in regular variables. We call this working with bitwise variables, because we can treat each bit as if it were its own variable.

To set or unset a bit in a variable, simply access the bit position with the {} braces like this...

```
playerflags{0}=1
enemyflags{3}=0
```

Bitwise access can also be used as a true or false condition if...then statements as well.

```
if playerflags{0} then goto playerdeath
```

Notice that we didn't test if playerflags{0}=1. Since the bit can already only be 1 or 0 – true or false – we don't check its value like we would with a regular variable.

We'll cover other aspects of if...then statements a bit later.

Note: many advanced statements in 7800basic don’t work directly with bitwise-arrays. To use bitwise-arrays with these statements, you will need to assign any bitwise-array lookups to a variable, and use the variable with those statements.

constant variables

A constant is a special type of variable that cannot be changed while a program is running. To declare a constant in 7800basic, use the const command. const declares a constant value for use within a program. This improves readability in a program in the case where a value is used several times but will not change, or you want to try different values in a program but don't want to change your code in several places first.

For example, you might have the following near the beginning of your program:

```
const MyConst = 200
const Monster_Height = $12
```

After that, any time MyConst or Monster_Height is used, the compiler will substitute 200 or $12 respectively.

indirect variable arrays

Indirect variable arrays are similar to regular variable arrays, but instead of accessing memory from a certain ROM or RAM location, they reference memory pointed to by a two-byte memory variable. (the two bytes of memory must be dimmed to consecutive a-z memory locations) This is typically used to access the advance POKEY sound registers, but you may also use indirect variable arrays if you wish to use the same code with different bits of data.

An indirect variable array is signified by using double square braces around the array index. To initialize the pointer to point to some data, you must use const to reference the data values...

```
dim mempointer=a
dim mempointerhi=b
dim memindex=c

rem **each data statement you wish to point at should have a similar
rem **const statement.
const mydataplo=#<mydata
const mydataphi=#>mydata

rem **set the memory pointer to point to “mydata”
mempointer=mydataplo
mempointerhi=mydataphi

rem **an example of indirect variable array access
if mempointer[[memindex]]=0 then goto drawelephant

data mydata
0,3,5,8
```

```
end
```

Indirect variable arrays work with variable assignments and if...then statements.

If your indirect array is accessing a data statement, you can forgo the constant definitions in the example above, and just add “_hi” and “_lo” to the data label name.

```
dim mempointer=a
dim mempointerhi=b
dim memindex=c

rem **set the memory pointer to point to “mydata”
mempointer=mydata_lo
mempointerhi=mydata_hi

rem **an example of indirect variable array access
if mempointer[[memindex]]=0 then goto drawelephant

data mydata
0,3,5,8
```

Note : many advanced statements in 7800basic don’t work directly with indirect arrays. To use indirect arrays with those statements, you will need to assign any indirect array lookups to a variable, and use the variable with those statements.

8.8 fixed point variables

Fixed point variables can be assigned fractional values, similar to floating point variables in other languages. 7800basic stores the integer part of the value in one byte, and the decimal part of the value in another.

To use fixed point variables you need to name them with dim, but this time we tell 7800basic that we'll be representing the variable with 2 parts.

```
dim playerx=j.k
```

After that, using fixed point variables is pretty much as easy as using regular variables, except you can add or subtract values with a decimal point. This comes in handy, for example, when you want to move an object at slower speed than the frame rate.

```
playerx=playerx+0.2
```

The decimal point aspect of fixed point numbers is limited to addition and subtraction. When it comes to other operations in 7800basic, like if...then statements, the language works with the variable by just referencing the “large” whole number part.

special variables

the score variables and BCD variables

7800basic provides you with two 24-bit score variables that you can use to track points for two player games, score0 and score1.

If you need more score variables than the two provided, you can dimension additional score2 to score9 variables, ensuring each dimension location has two free bytes of memory directly following it.

```
dim score2=g : rem ** h and i are also used for score2
```

Usage of the score variables is straightforward. You can set the score variables with a regular assignment.

```
score0=1000
```

And any time you wish to change the score, just do so with a regular addition or subtraction operation.

```
score0=score0+10
```

Please be aware that 7800basic cannot reliably add a variable to the score unless the variable is in Binary Coded Decimal format, or BCD for short. BCD is an alternate way for computers to represent decimal numbers in binary.

To assign numbers in BCD, you just take your decimal number you wish to assign, and add the hexadecimal $ specifier in front. For example, if we wanted a BCD variable to be 13, we'd assign $13 to it.

```
z=$13
```

When performing operations on non-score BCD format numbers, you need to use the “dec” command to let 7800basic know it should do its work in BCD format.

```
dec z=z+1
```

It should be noted that multiplication and division don't work on numbers in BCD format in 7800basic, due to lack of support in the underlying hardware.

If you wish to display a regular variable, 7800basic also includes a “converttobcd” utility function that returns the BCD value of a non-BCD variable. The non-BCD variable should be in the range between 0-99, or else it will start to display incorrect values.

```
BCDVar=converttobcd(NonBCDVar)
```

random numbers

rand is a special variable that will implicitly call a random number generator when used.

The rand function returns a pseudo-random number between 1 and 255 every time it is called. You typically call this function by something like this:

```
a = rand
```

However, you can also use it in an if...then statement:

```
if rand < 32 then r = r + 1
```

You can also set the rand variable to a specific value, at least until it is accessed again. The only reason you would ever want to do this is to seed the randomizer. If you do this, pay careful attention to the value you store there, since storing a zero in rand will "break" the randomizer, and all subsequent reads will also be zero!

temporary variables

There are 9 temporary variables, named temp1 through temp9, that are used by various 7800basic commands. You may reuse these variables so long as your code doesn't call one of the commands that overwrites it. (typically the plot functions, some cases of multiplication and division, and function calls)

the CARRY variable

The 6502 CPU sets something called “the carry flag” every time an addition results in a value that exceeds 255, and every time a subtraction results in a value less than 0.

7800basic allows you to check the status of this flag with if...then. This is particularly useful if you wish to perform events every N frames...

```
enemytick=enemytick+33
if CARRY then gosub enemymove
```

the pokeydetected variable

When your program has POKEY support enabled with “set pokeysupport on”, you can check to see if 7800basic detected a POKEY chip through the pokeydetected variable.

```
if pokeydetected then gosub playpokeymusic
```

the paldetected variable

You can check to see if 7800basic detected a PAL console through the paldetected variable.

```
  if !paldetected then frameskip=frameskip+213: if !CARRY then goto
skipmoveplayer
```

Conditional Logic

Perhaps the most important statement is the if-then statement. These can divert the flow of your program based on a condition. The basic syntax is:

if condition then action

action can be a statement, label or line number if you prefer. If the **condition** is true, then the statement will be executed. Specifying a line number or label will jump there if the **condition** is true. Put into numerical terms, the result of any comparison that equals a zero is false, with all other numbers being true.

There are three types of if-then statements.

1. simple true/false evaluation

The first type is a simple check where the condition is a single statement.

The following example diverts program flow to line 20 if a is anything except zero:

```
if a then 20
```

This type of if-then statement is more often used for checking the state of various read registers. For example, the joysticks, console switches, single bits and hardware collisions are all checked this way.

```
if joy0up then x = x + 1
```

That will add 1 to x if the left joystick is pushed up.

```
if switchreset then 200
```

Jumps to line 200 if the reset switch on the console is set.

```
if collision(player1,playfield) then t = 1
```

Sets t to 1 if player1 collides with the playfield.

```
if !a{3} then a{4} = 1
```

Sets bit 4 of a if bit 3 of a is zero.

2. simple comparison

A second type of statement includes a simple comparison. Valid comparisons are = , < , >, <=, >=, and <>.

```
if a < 2 then 50
if f = g then f = f + 1
```

```
if r <> e then r = e
```

3. compound statement

The third type of if-then is a complex or compound statement, that is, one containing a boolean AND (&&) operator or a boolean OR (||) operator. You are allowed only one OR (||) for each if-then statement. You can use more than one AND (&&) in a line, but you cannot mix AND (&&) and OR (||).

```
if x < 10 && x > 2 then b = b - 1
if !joy0up && gameover = 0 then gameoverhandler
if x = 5 || x = 6 then x = x – 4
```

Warning: Using multiple if-thens in a single line might not work correctly if you are using boolean operators.

Code Organization and Flow Control

bank

If you've chosen to use one of the bankswitch formats (see the “set romsize” command) you need to break up your game into banks. To signify any code that follows belongs to a particular bank number, you use the bank command...

```
[code belonging to bank 1]

  bank 2

[code belonging to bank 2]
```

You don't need to use the bank command to specify bank 1, as 7800basic assumes that a bankswitched game begins in bank 1.

dmahole

The 7800 requires it's graphics to be padded with zeroes. To avoid wasting ROM space with zeroes, 7800basic uses a 7800 feature called holey DMA. This allows you to stick program code in these areas between the graphics blocks that would otherwise be wasted with zeroes.

To direct 7800basic to store any code that follows in these areas, just use the dmahole command...

```
  dmahole 0
```

If you have multiple dmahole areas you can use multiple dmahole commands.

7800basic will stop directing code into the current dmahole when another dmahole command is

encountered, or if a new bank is declared.

7800basic will automatically add assembly code to allow your BASIC program to flow across a DMA hole. If you're creating a bankswitched ROM and intend to fill the last bank entirely with graphics, you'll need to disable that extra generated assembly code. You can do by adding the noflow keyword to the dmahole command...

```
dmahole 0 noflow
```

goto

To leave one section of code and go to another, you create a label above the destination code, and use the goto command. The destination code can be anywhere in the program, as can the goto.

```
  If enemyy>150 then goto blowupenemy
  [more program code here]

blowupenemy
  [more program code here]
```

If your game is bankswitched, you can add a bank destination to your goto command.

```
goto nukethemall bank4
```

gosub and return

To leave one section of code for another, but eventually return back, you use the gosub command. Gosub works almost the same as goto, except your destination area must eventually use the return command.

```
  If UFO=1 then gosub moveUFO
  [more program code here]

moveUFO
  [more program code here]
  return
```

If your game is bankswitched, you can add a destination bank number to your gosub command.

```
gosub aliensnot bank4
```

To return from a bankswitched gosub, use “return otherbank”.

If you have a routine that is called both locally and from other banks, you can end your routine with a “return” instead of “return otherbank”, and the bankswitch routines will sort out what to do.

If your game is bankswitched, you may want to use “return thisbank” instead of “return” for local returns, so 7800basic can use a bit less ROM space, and save a little bit of time too.

on...goto on...gosub

To goto or gosub to different destinations depending on the value of a variable, you can use on...goto or on...gosub.

```
  on mokeysize gosub smallmonkey mediummonky
  [more program code here]

smallmonkey
  rem we're here if monkeysize was equal to 0
  [more program code here]
  return

mediummonkey
  rem we're here if monkeysize was equal to 1
  [more program code here]
  return
```

Please note that 7800basic won't make sure that your variable isn't larger than the number of labels used in on...goto or on...gosub. If there aren't enough labels in on...goto to handle the variable, your code will likely do unexpected things, up to and including crashing the console.

loadrombank

When using bankswitching, there may be times you wish to switch the active bank for using its data without actually calling goto or gosub. You may do this from the last always-present bank.

```
loadrombank bank2
```

loadrambank

When using bankswitching with the BANKRAM formats, you can switch the active RAM bank with the loadrambank command.

```
loadrambank bank1
```

reboot

The reboot command will restart your game as if the console had just been turned on. It uses no arguments...

```
reboot
```

for...next

For...Next loops work similar to the way they work in other Basics.

The syntax is:

for variable = value1 to value2 [step value3]

variable is any variable, and **value1**, **2**, and **3** can be variables or numbers. You may also specify a negative step for **value3**.

The step keyword is optional. Omitting it will default the step to 1.

```
for x = 1 to 10
for a = b to c step d
for l = player0y to 0 step -1
```

Normally, you would place a variable after the next keyword, but 7800basic ignores the keyword and instead finds the nearest for and jumps back there. Therefore, the usual way to call next is without a variable. If any variable is specified after a next, it will be ignored.

```
for x = 1 to 10 : a[x] = x : next
```

It is also important to note that the next doesn't care about the program flow — it will instead find the nearest preceding for based on distance.

```
   for x = 1 to 20
   goto skipout
   for g = 2 to 49
skipout
   next
```

The next above WILL NOT jump back to the first for, instead it will jump to the nearest one, even if this statement has never been executed. Therefore, you should be very careful when using next.

user functions

A simple interface is provided for you to define your own functions in 7800basic. These functions can be defined within your program itself or compiled to their own separate .asm file then included with the include command. Functions can be written in basic or assembly language.

To call a function, you assign it to a variable. This is currently the only way to call a function. Functions can have up to six input arguments, but they only have one explicit return value (that which is passed to the variable you assigned to the function.) You can have multiple return values but they will be implicit, meaning that the function will modify a variable and then you can access that variable after you call the function.

A function should have input arguments. In 7800basic, a function can be called with no input arguments if you want, but you might as well use a subroutine instead, as it will save space.

To declare a function, use the function command, and specify a name for the function. Then place your basic code below and end it by specifying end. To return a value, use the return keyword. Using return without a value will return an unpredictable value.

Note that in 7800basic, all variables are global, and arguments are passed to the function by use of the temporary variables temp1-temp6. Therefore it is recommended that you use the same temp variables for calculations within your function wherever possible so that the normal variables are not affected.

Example of declaring a function in 7800basic:

```
   function sgn
   rem this function returns the sign of a number
   rem if 0 to 127, it returns 1
   rem if -128 to 1 (or 128 to 255), it returns -1 (or 255)
   rem if 0, it returns 0
   if temp1=0 then return 0
   if temp1 <128 then return 1 else return 255
end
```

To call the above function, assign it to a variable, as follows:

```
   a = sgn(f)
```

Note that there is no checking to see if the number of arguments is correct. If you specify too many, the additional arguments will be ignored. If you specify too few, you will likely get incorrect results.

Data

For convenience, you may specify a list of values that will essentially create a read-only array in ROM. We'll go over the different commands in 7800basic to accomplish this.

regular data

You create these lists of values as data tables using the data statement. Although the data statement is similar in its method of operation as in other Basic languages, there are some important differences. Most notably, access to the data does not need to be linear as with the read function in other Basics, and the size is limited to 256 bytes.

If you prefer to use a data statement similar to that in other Basics and don't want to be limited to 256 bytes, see Sequential Data below.

In a regular data statement, any element of the data statement can be accessed at any time. In this vein, it operates like an array. To declare a set of data, use data at the beginning of a line, then include an identifier after the statement. The actual data is included after a linefeed and can continue for a number of lines before being terminated by end. Suppose you declare a data statement as follows, with array name _My_Data:

```
   data _My_Data
   200, 43, 33, 93, 255, 54, 22
end
```

To access the elements of the data table, simply index it like you would an array in RAM. For example, _My_Data[0] is 200, _My_Data[1] is 43, ... and _My_Data[6] is 22. You can only use this method to retrieve up to 256 elements, since the index is byte sized.

To help prevent the reading of values beyond data tables, a constant is defined with every data statement — this constant contains the length, or the number of elements, in the data. The constant will have the same name as the name of the data statement, but it will have _length appended to it.

For example, if you declare:

```
   data _My_Data
   1,2,3,4,5,6,7,8,9
end

      you can then access the length of the data with _My_Data_length. You can
      assign this to variables or use anywhere else you would use a number.

         a = _My_Data_length
```

These data _length constants will not work correctly if they are used before you declare the corresponding data statement. If you need to use a data _length constant before its data statement, declare the data _length constant near the beginning of your program (using the name of the constant as the value).

```
         const _My_Data_length=_My_Data_length
```

Note again that these data tables are in ROM. Attempting to write values to data tables with commands like _My_Data[1]=200 will compile but will perform no function.

sequential data

The sdata statement will define a set of data that is accessed more like the data in other Basics. The 256-byte limitation is also removed. Sequential data is useful for things like music or a large set of scrolled playfield data. There is also no need to specify a pointer into the data.

Sequential data statement requires two adjacent variables to be set aside.

To define the set of data, use:

sdata <name>=<variable>

<name> is the name you wish to call it when it is read, and <variable> is the first variable name you are setting aside. Although you just specify one variable, the next variable in sequence will also be used.

```
         sdata _My_Music=a
         1,2,3,4,5,6,7,255
      end
```

The above will set up a sequential data table called _My_Music that uses variables a and b to remember the element at which it is currently pointing.

Unlike regular data statements, the program must actually run over the sdata statement for it to be properly initialized to the beginning of the data table. Also, it must typically be defined outside the game loop because each time the program runs over the statement, it will be reinitialized to the beginning of the table.

To read the data, use the sread function. It works somewhat similar to a regular 7800basic function.

```
t = sread(_My_Music)
```

This will get the next value in the data table _My_Music and place it in t.

Note that unlike other Basics, there is no error or other indication when reaching the end of data. Since there is no data length variable defined with sequential data, it's usually necessary to place a terminal value at the end of your data. In the above example, 255 was used. In the above example you would then check t for 255.

There is no restore function like other basics. However, if you allow your program to run over the sdata statement again, it will be initialized to the beginning of data just like the restore function.

using character data with alphachars and alphadata

To help with creation of tile maps and text strings, 7800basic has special data functions to reference each graphical character.

If you wanted to create text data that would be used to draw characters from a graphic file called “tileset_chars.png” that looks like this...

...you would first give 7800basic a heads-up as to which letters you wish to use to represent these graphics with the alphachars command.

```
alphachars ' abcd'
```

7800basic doesn't “know” the letters A, B, C. Instead, your alphachars statement tells 7800basic that any alphadata commands that follow should represent the space, A, B, C, and D, as the 0th, 1st, 2nd, and 3rd characters in the image file.

Once you've defined the characters in the image file with alphachars, you may go ahead and use them in one or more alphadata commands.

```
 alphadata mytext tileset_chars
 'bad dad'
 'abacab '
end
```

Plotting the characters on the screen would be just a matter of calling plotchars with the “mytext” data.

Recall from the chart in the Graphics Modes section that most 7800modes represent a character with less than 8 pixels of width. If you would prefer to use wider character graphics than your mode allows, you have 2 choices.

1. You can set MARIA to doublewidth mode with the command...

```
 set doublewide on
```

This will make every character use two bytes instead of one, doubling the pixel width of each character.

2. If you only want to display some tiles with extra width and others with the normal character width, you can use the “extrawide” parameter with alphadata.

```
alphadata testtext alphabet_8_wide extrawide
 'copyright'
end
```

Alphadata can also be used to help entry of non-alphabet data.

If we had a tile graphic that looks like this...

...you could use the following alphachars to give a letter to each tile.

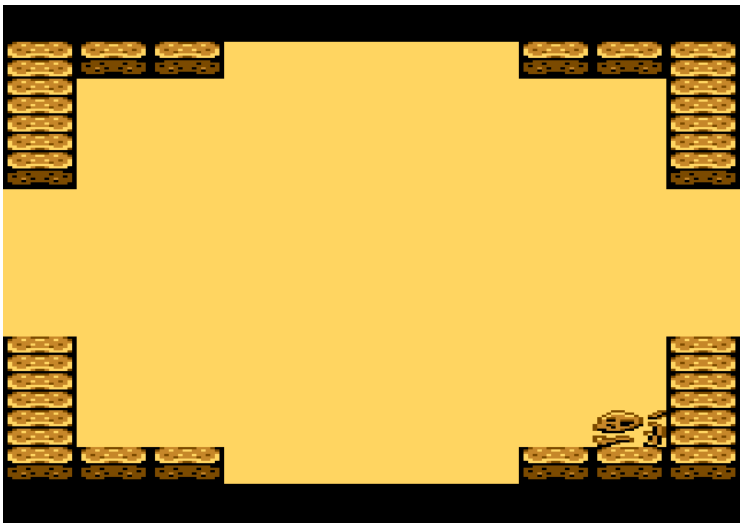
```
 alphachars ' abcdefg'
```

Which would tell 7800basic to reference tiles in the graphic with the following letters...

After which we can define a screen with alphadata...

```
 alphadata screen1map tileset_blanks
 'fgdede        dedefg'
 'fg                fg'
 'fg                fg'
 'de                de'
 '                    '
 '                    '
 '                    '
 '                    '
 'fg                fg'
 'fg                fg'
 'fg              bcfg'
 'dedede        dedede'
end
```

Which which, we could use the data with plotmap to display the screen.



There are few more details we need to cover on using the alphachars command.

The default alphachars definition is ' abcdefghijklmnopqrstuvwxyz.!?,”$():'. This matches the graphical content of the sample *alphabet_4_wide.png* and *alphabet_8_wide.png* files that were provided with 7800basic.

If you wish to work with a full ASCII characterset, like the ATASCII set provided with 7800basic, you can quickly setup alphachars with “alphachars ASCII”.

If you wish to work with the high score font characterset, you can quickly setup alphachars with “alphachars HISCORE”. You can reference the copyright symbol with the capital “C” letter in your text.

If you have changed alphachars in one part of your code and wish to go back to the default set later, you can use the command “alphachars default”.

Tiled character-map data import

In the previous section we took a look at using alphadata to represent character maps. There is another, more flexible, way to represent character maps in 7800basic, and that's using the incmapfile command.

incmapfile allows you to import tile layouts that you have created in the Tiled graphical editor. The Tiled TMX files should be in XML format (under Edit->Preferences), and in Tiled you should use the same png image files you are using with incgraphic in your 7800basic program.

Once you've created the screen, import is just a matter of issuing the incmapfile command with the filename.

```
incmapfile screen1map.tmx
```

You may then use the map name with the plotmap command, using the filename without the extension as the “mapdata” parameter.

It should be noted that you may only use characters in the filename that would be valid for 7800basic labels, so stick to alphanumeric characters, and avoid spaces and characters that have other meanings.

speechdata

Speechdata is a command used to generate voice data for use with the speak command, which sends data to the AtariVox speech module. Like other data commands, speechdata must be terminated by the end keyword.

```
  speechdata downspeech
  reset
  phonetic   'may the force be with yoo.'
end
```

The data itself contains several keywords, each of which represent the different types of data that AtariVox will accept. In the example above, you see reset data, followed by phonetic speech data .

Here's the list of valid data types for use in the speechdata command.

reset
Causes AtariVox to reset current speech and parameters

pitch #
Any future speech data will use the provided pitch

speed #
Any future speech data will use the provided speed.

raw #,#,#,...

The raw data provided is sent to the AtariVox. This is useful if you wish to send more complex commands to the AtariVox. See the SpeakJet manual for more information.

dictionary 'word phrase'
Each word is converted to AtariVox data via 7800basic's dictionary. Any words that aren't in the dictionary are run through phonetic translation.

phonetic 'word phrase'
Each word is converted to AtariVox data via 7800basic's phonetic translation. You will likely need to misspell complex words to get the pronounciation you desire, though first give the correct spelling a try before you try tweaking it.

For both dictionary and phonetic data, if you use a period at the end of a word, it will drop the pitch partway through the word and invoke a pause. Similarly, a question mark will cause a rise in pitch and similar pause. Spaces and commas will provide moments of silence.

Program Configuration

7800basic has a series of “set” commands you may use to customize the 7800basic environment configuration. These should be at the top of your program, prior to other commands.

set romsize [value]

Ths sets the ROM size and format of your game. The default is 32k, and valid values for the romsize are 16k, 32k, 48k, 128k, 128kRAM, 128kBANKRAM, 144k, 256k, 256kRAM, 256kBANKRAM, 272k, 512k, 512kRAM, 512BANKRAM, and 528k.

```
set romsize 128k
```

Formats larger than 48k are bankswitching formats, with many 16k banks. While you need to bankswitch to access all banks except the last, which is always present. If you use bankswitching format, you'll need to use the bankswitching goto/gosub commands to move between different banks.

The formats with RAM on the end of the name provide an extra 16k of RAM from $4000-$7fff. You can access this extra RAM by naming it with dim, and using the memory through regular variable access.

```
dim treasuremap=$4000
```

The formats with BANKRAM on the end of the name provide 2 banks of 16k RAM from $4000-$7fff. You may switch the active RAM bank with the loadrambank command.

The formats 144k, 272k, and 528k, all have bank 1 permanently available at the $4000 address space. This means you don’t have to bankswitch to reach bank 1 with these formats. Similarly, you don’t need to bankswitch to ensure graphics in bank 1 are displayed.

set doublewide on

This tells MARIA to fetch 2 bytes of character data for each character you plot, effectively making the characters twice as wide.

set zoneheight [value]

The default zone height is 16, but can be defined once in your program with the zone height setting.

```
set zoneheight 8
```

There are more details on zone heights in the *zoneheight* entry in the *Graphics Commands* section of this guide.

set tallsprite [on|off|spritesheet]

If the tallsprite is set to “on” (the default) then when the 7800basic incgraphic command sees a graphic that is 2 zone-heights or more, it will import all of the graphics and link them together as a “tallsprite”. If you use the plotsprite command on a tallsprite graphic, the plotsprite command will loop to ensure all parts will be drawn.

If you wish to force incgraphic to only import 1 zone-height’s worth of graphic data, you can issue the “`set tallsprite off`” command.

If you wish to import a png with a series of zone-sized vertical sprites, you can “`set tallsprite spritesheet`”. This has the benefit of ensuring all of the index colors are consistent between frames. The one drawback is the resulting sprites aren’t uniquely named, and need to be drawn with plotsprite command [frame] parameter.

set screenheight

The default screen height is 192, but it can be defined once in your program with the screen height setting. Valid height values are 192, 208, and 224.

```
set screenheight 208
```

There are more details on screen heights in the *screenheight* entry in the *Graphics Commands* section of this guide.

set extradlmemory on

This setting tells 7800basic that you want to use the memory at $2200 for the purpose of expanding your display list, to allow more objects. When you compile your program with this option, the compile output will tell you what memory values it used.

```
$2250 to $27ff was claimed as extra DL memory.
```

In this example, you’re still able to use the memory from $2200 to $224F for your program.

This option is incompatible with set dlmemory.

set dlmemory [first byte location] [end byte location]

By default, 7800basic allocates it’s object display list from $1880 to $1fff. If you have another source of available memory (like on-cart memory) and you need to increase the number of objects that the display can hold, you may wish to relocate the display list.

```
set dlmemory $4000 $4fff
```

You can see the number of possible display objects during the game compilation.

```
7800basic compilation complete.
User-defined 7800.asm found in current directory
   7505 bytes of ROM space left in the main area of bank 1.
         4096 bytes of ROM space left in DMA hole 0.
   $18c0 to $1fff used as zone memory, allowing 14 display objects per zone.
   2314 bytes left in the 7800basic reserved area.
```

Note: Even though you can increase the number of bytes of display storage, this still isn’t a guarantee you will have time to actually plot or display all those objects. Check out the *doublebuffer* command if you need more than one frame to build your display.

set plotvalueonscreen [on|off]

This tells 7800basic that when the plotvalue command is used it should update the screen immediately, instead of waiting for the screen to complete drawing. This will save precious off-screen cycles for plotting other objects, like sprites.

When using this configuration option, a program should issue its plotvalue commands in the same order, immediately after the restorescreen or clearscreen commands. This will ensure they go into the display list in the same order as previous frames, which prevents glitching.

set plotvaluepage [$##]

The plotvalue function converts the variable you pass to it into character-set codes. These character codes take up room in a buffer, which limits the total number of plotvalue characters to 64. This should be enough for most game types, but if your game has a screen full of values the 64 character limit won’t be enough. You can provide your own 256-byte storage page with the `set plotvaluepage` command. This will allow for 256 characters of value display.

```
set plotvaluepage $27
```

set zoneprotection [on|off]

If too many objects are plotted to the same horizontal zone, there's a chance objects will overflow the storage for that zone, and corrupt display of the next zone. Using this command will prevent that from happening, at the expense of available CPU time.

It's unlikely this protection is needed when the zoneheight is 16. It may be needed if the zoneheight is 8 and the game has many free roaming objects.

set pauseroutine [on|off]

Turning pauseroutine off tells 7800basic to not use its built-in pause routine in the game.

set paddlerange [#]

This sets the value range the paddle controllers will return. Larger ranges use more cpu time.

set paddlescalex2 [on|off]

This doubles the values returned by the paddles, increasing the range without additional cpu time.

set paddlepair [on|off]

Tells 7800basic to read both paddles in a set of 2.

set mousexonly [on|off]

Tells 7800basic to only read the X axis of mice, to conserve cpu time.

set drivingboost [on|off]

Tells 7800basic to apply a boost to values, when the driving wheel controller is turned quickly.

set pokeysound [on|off]

This tells 7800basic to look for and configure a POKEY chip. There are more details in the *POKEY Sound* section of this guide.

set trackersupport basic

This tells 7800basic to include the music tracker in your game. For more details see the *Music Tracker* section of this guide.

set tiasfx mono

This tells 7800basic to only use one channel for TIA sound effects. For more details see the *playsfx* command documentation.

set avoxvoice on

This tells 7800basic to turn on AtariVox voice support. For more details see the *speak* and *speechdata* command documentation.

set debug color

This tells 7800basic to change the color of the screen to red, until all of the screen plotting functions are called. This allows you to judge how much of the program calculations are happening during the visible screen, and allows you to see the relative efficiency of your program.

set canary [on|off]

7800basic checks a memory location called “the canary” to see if it’s stack memory is being overwritten by your game. If this location changes, the screen will turn red and the game will stop. By default the canary protection is on, but you can use this statement to disable the protection.

set mcpdevcart [on|off]

This tells 7800basic to use hotspots compatible with the 7800 MCP DevCart. This only needs to be used when working with bankswitched formats.

set basepath [directory path]

This tells 7800basic to add the provided directory path to any relative paths used with the incgraphic, incmapfile, and plotmapfile commands.

set hssupport $####

This tells 7800basic to include support for saving to the Atari High Score Cart, AtariVox, and Savekey. The #### is a unique hex number that identifies your game. You can reserve by announcing it in this AtariAge thread: http://atariage.com/forums/topic/128432-high-score-cart-values/

set hsseconds

This allows you to control how many seconds the “drawhiscore attract” command will display for, before continuing on to the basic command.

set hsscoresize

This determines how many digits will be displayed for scores displayed by the “drawhiscore” command.

set hscolorbase

This determines which color value the color cycling in the high score table will begin with.

set hsdifficultytext off

This turns off the difficulty level names in the high score display, which is useful for games with only one difficulty level.

set hsdifficultytext 'diff string 1' 'diff string 2' 'diffstring 3' 'diff string 4'

This allows you to provide custom names for the difficulty levels, instead accepting the default of *easy*, *intermediate*, *advanced*, and *expert*. The single quotes are only necessary if your names include spaces.

set hsgameranks # 'rank string 1' # 'rank name 2' # 'rank name 3' [...]

If you use this parameter, the high score tables will display a descriptive ranking of player's score. The scores+descriptions should be listed in descending order, with the last score listed being 0. The single quotes are only necessary if your descriptions include spaces.

```
set hsgameranks 30000 commander 10000 lieutenant 0 ensign
```

Graphics Commands

displaymode

This command sets the current graphics display mode. You may choose between 160A, 320A, 320B. It should be noted that these modes are also capable of displaying 160B, 320C, and 320D formatted graphics, respectively.

```
displaymode 320A
```

clearscreen

This command erases all sprites and characters that you've previously drawn on the screen, so you can draw the next screen.

savescreen and restorescreen

The savescreen command saves any sprites and characters that you've drawn on the screen since the last clearscreen. The restorescreen erases any sprites and characters that you've drawn on the screen since the last savescreen.

These commands are meant to reduce the CPU requirements of your game, by avoiding re-plotting background elements that don't change from frame to frame.

There are a couple of restrictions worth noting.

1. You must use restorescreen with savescreen. If the clearscreen command is issued, the last saved screen is lost.
2. Only one screen is ever saved. ie. if you save the screen many times, you can't keep restoring many times to get back to the first screen.

drawscreen

The drawscreen command ensures that all plotted graphics are ready to display, and waits for MARIA to start the display. Including drawscreen in any display loop ensures that the loop will run every 1/60th of a second for NTSC consoles, or 1/50th of a second for PAL consoles.

drawwait

The drawscreen command completes near the beginning of the visible display. This is done intentionally, to allow your program to have the maximum amount of CPU time possible.

You may occasionally have code that you don't want to execute during the visible screen. For these

occasions you can call the drawwait command. This command will only return after the visible screen has been completely displayed.

It should be noted that the plot* commands do the equivalent of calling drawwait for you, since modifying the screen display while it is being drawn would cause display glitches.

working with the screen commands

Here's one example of putting the previous screen commands to work...

```
LoadLevel
  clearscreen
  gosub DrawBackgroundMap
  gosub DrawScoreBackdrop
  savescreen
main
  restorescreen
  [game logic and sprite plotting]
  drawscreen
  goto main
```

Though games with ever-changing backgrounds or non-character single-color backgrounds may choose to simply draw everything single thing every single frame, since they don't benefit from saving the background...

```
main
  clearscreen
  [game logic, character and sprite plotting, score display]
  drawscreen
  goto main
```

doublebuffer

If your game takes a very long time to build the display, you may wish to use the double buffering system in 7800basic, which frees you from the requirement of completing a display within a single frame. The double buffering system is controlled through the doublebuffer command.

```
doublebuffer [on|off|flip] [minimum framerate]
```

When using double buffering, you don’t need to use drawscreen anymore. The program flow will look something like this.

```
  doublebuffer on
  clearscreen
  [plot* commands to draw background]
  savescreen
mainloop
```

```
restorescreen
[game logic, character and sprite plotting, etc.]
doublebuffer flip
goto mainloop
```

You can optionally set a minimum framerate with the doublebuffer command, to ensure there's a uniform framerate when your game logic and object count differs.

```
doublebuffer on 2
```

Note: Double buffering requires two object buffers, so it will reduce the number of objects any given zone can display. To learn how to increase the number of objects, see the *set dlmemory* command.

the topscreenroutine subroutine

If your game has a subroutine named "topscreenroutine", this routine will be called at the beginning of each visible frame. This is useful if you wish to change the screen display as it's being drawn, changing the background color, changing the display mode, etc.

You shouldn't modify the 7800basic temp1-temp9 variables, or any other key game variables in this routine. If you need temporary variables, you may use the inttemp1-inttemp6 variables.

See the *splitmodedemo* in the 7800basic *samples* directory for more information.

setting palettes

MARIA has 8x 3-color palettes to choose from when drawing sprites and characters. Setting the actual color these palettes use is just a matter of setting some special variables.

```
rem ** set palette 0 colors to green, light green, and salmon
P0C1 = $d2
P0C2 = $d8
P0C3 = $3b

rem ** set palette 1 colors to grey, white, and yellow
P1C1 = $04
P1C2 = $0f
P1C3 = $18
```

zoneheight

Graphics in 7800basic are limited to either 8 or 16 pixels tall. This is a result of MARIA's zone based architecture. To use taller sprites in your game, simple define more sprites and position one above the other. Other 7800 games with sprites taller than a zone are doing the same, one way or another.

The default zone height is 16, but can be defined just once in your program with the zone height setting.

```
set zoneheight 8
```

Using a zone height of 8 means that 7800basic needs to present more memory to MARIA for screen building. As a consequence, when you use a zone height of 8 you may only display up to 16 objects within any zone. When you use a zone height of 16, you may display up to 20 objects on any zone.

screenheight

The vertical resolution of your screen in 7800basic defaults to 192 pixels, but you may set it to 208 or 224 pixels with “set screenheight”.

```
set screenheight 208
```

The drawback of setting the screen height taller than 192 is it reduces the number of off-screen cycles your game may use. This may impact the number of objects you can plot on the screen, if your game has a large number of moving objects.

adjustvisible

If the very top or bottom part of your game screen doesn't have moving objects in it, you can use the adjustvisible command to tell 7800basic where the area that recieves object updates is. This will make more CPU cycles available to your game.

The adjustvisible command takes the top visible zone and bottom visible zone as an argument. For example, if you wanted to exclude the top 2 zones in a game with zoneheight set to 8, the following command would do so.

```
adjustvisible 2 23
```

The default values for a game with a zoneheight of 8 and screenheight of 192 would be 0 and 23. The default values for a game with a zoneheight of 16 and a screenheight of 192 would be 0 and 11.

Be warned that if you use adjustvisible and continually plot objects into the “non-visible” area, there will likely be some display glitching for those objects.

incgraphic

Being a command line program, 7800basic has no built-in image editor. Instead it has the ability to import standard .png format graphics.

When you create graphics, its important that you create them to adhere to the 7800 mode you want to import them in. If you have too many colors in your image, or if it the image has a width that isn't a multiple of the mode's character width, the import will fail with a compile error.

Images should also be in “indexed” format, rather than “rgb” format.

Whether you're importing a tileset or sprite image, the same incgraphic command is used.

```
incgraphic filename.png [graphics mode] [color #0] [#1]  ... [palette #]
```

The graphics mode is the name of the various MARIA display modes, 160A, 160B, 320A, 320B, 320C, or 320D. If you don't enter a mode, your image will default to being imported for 160A mode.

Since your image editor probably doesn't provide control over which color uses which index, 7800basic provides the ability to change which png color index goes to which MARIA color index. Here's an example of swapping the second and third colors.

```
incgraphic foobar1.png 160A 0 2 1 3
```

The palette parameter needs to be defined if you plan to use your image as part of a map to be displayed with the plotmapfile command. It tells plotmapfile which palette number it should use when drawing any tile from the image.

The graphic plotting commands will access your image by it's filename, without the .png extension. Because of this it has certain restrictions on the file naming to avoid confusing the 7800basic compiler.

- each name should be unique.
- each name should contain only letters and digits.
- each name should begin with a letter character.

The png images should reside in the same directory as your source files, or in a subdirectory in the same directory as your source files. It's recommended that you use the subdirectory approach, since the number of graphic files used in a typical game can get overwhelming. Here's an example of using a subdirectory...

```
incgraphic gfx/foobar2.png
```

You may use Windows style directory separator slashes (“\”) or Unix style directory separator slashes. (“/”) Either style will work with 7800basic running on any platform.

When importing a graphic, incgraphic also creates “color constants” that you can use to set the palette entries for the sprite. Here's an example for setting palette 0 for the graphic “foobar3.png”.

```
P0C1 = foobar3_color1
P0C2 = foobar3_color2
P0C3 = foobar3_color3
```

Since 7800basic only learns of these values during the incgraphic command, you may only use these constants in your source code lines after the incgraphic command for the sprite you wish to set the color for.

If you wish to fine-tune the color that 7800basic has provided, you may add or subtract a constant value from it.

```
P0C2 = foobar3_color2 + $10
```

Note: For smooth glitch-free vertical movement with sprites, sprite graphics need to be placed into

memory between $9000 and $EFFF. This is due to Maria not implementing the DMA hole feature at memory locations below $8000.

You can check where your graphics are presently imported by looking at the messages that 7800basic puts out when it compiles your game.

If you desperately need to use the areas below $9000 for sprite graphics, you'll need to ensure adjacent areas to your sprite graphics there have no code or data in them. At a high level, you can do this by using the "dmahole #" command to ensure your code doesn't spill into those areas. Exact details here are very specific to your game layout and zoneheight choices, and are beyond the scope of this document.

incbanner

Incbanner is identical to the incgraphic command, but is used to import png graphics that are taller than a single zone's height.

```
incbanner filename.png [graphics mode] [color #0] [#1]  ...
```

newblock

You can use the newblock command to tell 7800basic to close off the current graphics block and begin a new one. This can be useful if you want to group certain graphics together in the same block. e.g. for sprite animations.

The command is used without arguments...

```
newblock
```

plotsprite

To display a sprite on the screen you use the plotsprite command.

```
plotsprite sprite_graphic palette_# x y [frame] [tallheight]
```

Where...

sprite_graphic is the the name of the imported graphic image you wish to display.

palette_# is the palette color set you wish the sprite to be drawn with.

x is the x screen coordinate of the new sprite.

y is the y screen coordinate of the new sprite.

[frame] is an optional parameter. If you have a series of sprites you wish to display (e.g. for a walk cycle) you can do so by using a variable here. If the value is 0 the first sprite will be displayed, if the

value is 1 the second sprite will be displayed, and so on. This feature requires all of the sprites to be the same width.

[tallheight] is the number of zones each tallsprite occupies, if you're using tallsprites. This parameter will make the frame parameter give correct results for tallsprites.

The display order of sprites and characters depends on the order they were drawn in. The very last sprite will be the one that appears to pass over top of other sprites.

plotbanner

To display a banner that was imported with incbanner on the screen, you use the plotbanner command.

```
plotbanner banner_graphic palette_# x y
```

Where...

banner_graphic is the the name of the imported banner graphic image you wish to display.

palette_# is the palette color set you wish the sprite to be drawn with.

x is the x screen coordinate of the new sprite.

y is the y screen coordinate of the new sprite.

It should be mentioned that in the 7800 architecture, to accomplish plotting a banner "N" zones tall, 7800basic actually plots "N" sprites. This should be taken into consideration for games that are already be pushing the sprite limites.

notes on use of characters

Character graphics in 7800basic have a few wrinkles you should keep in mind.

- Characters being used for any one frame of graphics must come from the same graphics area. The area will be 2k or 4k, depending on if you used 8 or 16 for your zone height setting.
- Characters can only be plotted on distinct lines. The Y coordinates in character functions refers to a coarse/line position, not the screen Y coordinate.
- MARIA can only plot 32 characters across with a single command. If you need more than 32 characters across, you'll either need to draw all the characters in 2 commands, or you can set doublewide on to double the amount of graphics one character displays.

characterset

MARIA is only capable of accessing characters from one graphics block at a time. To choose the active graphics block containing your characters, you use the characterset command. As an argument you just

provide the name of one of the character graphics in the block you wish to activate.

```
characterset playfield_tiles
```

If you have multiple graphics blocks and wish to maximize the number of characters you can display in any one frame, group the incgraphic commands for all chacter graphics together so they wind up in the same block.

plotchars

To display a character or line of characters on the screen you use the plotchars command.

```
plotchars textdata palette_# x y [number_of_chars | extrawide]
```

Where...

textdata is the RAM, ROM, or literal text string you're trying to plot on the screen.

palette_# is the palette color set you wish the charters to be drawn with.

x is the x screen coordinate of the new characters.

y is the y line coordinate of the new characters.

number_of_chars is the number of characters you wish to draw. This isn't used if you've used a literal string for the textdata parameter.

extrawide can be specified if the graphics are twice the width of the character size. This optional parameter is only available if you've used a literal string for the textdata parameter. If you wish to make RAM or ROM character based data extrawide, the alphadata statement has a parameter for that.

```
plotchars 'Hello World!' 0 0 8
```

NOTE: If you are using a literal text string with plotchars, you must specify the graphic with your font, using the characterset command.

```
characterset alphabet_8_wide
```

plotmap

To display two or more rows of characters on the screen you may use the plotmap command.

```
plotmap mapdata palette_# x y width height [map_x_off map_y_off map_width]
```

Where...

mapdata is the RAM or ROM location of the data you're trying to plot.

palette_# is the palette color set you wish the charters to be drawn with.

x is the x screen coordinate of the new characters.

y is the y line coordinate of the new characters.

width is the number of columns of characters you wish to plot.

height is the number of lines of rows of characters you wish to plot.

The parameters that follow are optional, providing the ability to plot a small area of a larger data map definition.

map_off_x is the x coordinate of the area within the larger map

map_off_y is the y coordinate of the area within the larger map

map_width is the actual row width of the larger map

plotmapfile

If you've drawn a screen map using Tiled (see incmapfile) you may wish to use plotmapfile to display it instead of plotmap. Plotmap displays its data in a single palette, while using plotmapfile will allow your display to use different palettes in different parts of the screen. Also, plotmap is limited to displaying a maximum width of 32 characters, but plotmapfile will allow you to fill the display with characters.

```
plotmapfile mapfile.tmx mapdata x y width height
```

Where...

mapfile.tmx is the mapfile you wish to use as the screen map definition.

mapdata is the RAM or ROM location of the data you're trying to plot. In the case of ROM, this parameter will be the name of your mapfile without the extension.

x is the x screen coordinate of the new characters.

y is the y line coordinate of the new characters.

width is the number of columns of characters you wish to plot.

height is the number of lines of rows of characters you wish to plot.

It's important to note that when you import images for use with plotmapfile, you need to define their

“palette” parameter, so plotmapfile will know which palette to use for areas of the screen that use that image. See the plotmapfile entry for more information.

plotvalue

To display the value of a score or other BCD variable on the screen you use the plotvalue command.

```
plotvalue digit_gfx palette_# variable #_of_digits x y [extrawide]
```

Where...

digit_gfx is the png graphic containing the numeric characters.

palette_# is the palette color set you wish the charters to be drawn with.

variable is the variable containing the BCD value you wish to display on the screen.

#_of_digits is the number of digits you wish to display. If you specify more digits than 2, which is the maximum a BCD value can hold, the next memory location will be used for the next 2 digits, and so on.

x is the x screen coordinate of the new characters.

y is the y line coordinate of the new characters.

extrawide is an optional parameter. If use, plotval will display digits at 2x the normal character width. The png graphic containing the numeric characters will need to be designed at 2x the normal character width for this feature to work.

Note: because plotvalue uses an internal buffer to convert numbers to character strings, you can only plot up to 32 double-width or 64 single-width digits on the screen with plotvalue. If you need a larger number of values, please see the section for `set plotvaluepage`.

peekchar

The peekchar command is used to look up what value character is at a particular character position, in a character map. You may need to do this to, for example, to check if there's an item for the game's hero to eat, or to see if the game's hero is standing on solid ground.

peekchar may be used for RAM or ROM based maps.

```
charvalue=peekchar( mapdata, x, y, width, height )
```

Where...

mapdata is either a ROM data statement that you defined with alphadata, or a RAM memory area that you defined with dim.

x and y are the column and row where the you wish to check is.

width and height are the total width and height of the map. These should be actual values, rather than variables.

Note: the character index returned by peekchar is the byte-position of the character within the current characterset. If you've turned on double-wide characters, the 7800 still uses byte-position for these indexes, so your first character will be at index 0, then second will at index 2, the third at index 4, and so on.

pokechar

The pokechar command is used to set a character value within a RAM based character map. You may need to use this, for example, if the game's hero has touched a "food" character and you wish to make it vanish.

```
pokechar mapdata x y width height value
```

Where...

mapdata is a RAM memory area that you defined with dim.

x and y are the column and row where the you wish to check is.

width and height are the total width and height of the map. These should be actual values, rather than variables.

value is the character number you wish to store at the position.

Note: the character index used by pokechar is the byte-position of the character within the current characterset. If you've turned on double-wide characters, the 7800 still uses byte-position for these indexes, so your first character will be at index 0, then second will at index 2, the third at index 4, and so on.

lockzone

The lockzone command tells 7800basic to ignore any attempts to add any new sprites or tiles to a particular zone.

```
lockzone zonenumber
```

Where zonenumber is the zone you wish to lock. The first zone at the top of the screen is number 0.

Note: the lock/unlock state of zones is saved with the savescreen command, restored with the restorescreen command, and cleared with the clearscreen command.

unlockzone

The unlockzone command tells 7800basic to remove a zone lock that was added previously with the lockzone command.

```
unlockzone zonenumber
```

Where zonenumber is the zone you wish to lock. The first zone at the top of the screen is number 0.

Note: the lock/unlock state of zones is saved with the savescreen command, restored with the restorescreen command, and cleared with the clearscreen command.

shakescreen

The shakescreen command tells 7800basic to move the screen randomly, for visual effect.

```
shakescreen amount
```

Where amount is one of the following words: `hi`, `med`, `lo`, or `off`.

The shakescreen command needs to be called once per frame, to continuously move the screen. When you’re done with the shaking effect, you should call “shakescreen off” to restore the correct screen position.

memcpy

While it's not strictly a graphical command, the memcpy command is useful for copying ROM based character map data to RAM.

```
memcpy destination source number_of_bytes
```

Where...

destination is a RAM based memory area that you defined with dim.

source is a ROM data statement that you defined with alphadata.

number_of_bytes is the number of bytes you wish to copy. This can range from 1 to 65535, but must be an actual value, rather than a variable.

memset

Also not strictly a graphical command, but memset is often usefull for initializing RAM based map data.

```
memset destination value number_of_bytes
```

Where...

destination is a RAM based memory area that you defined with dim.

value is the value you wish to initialize each byte with. You can also use an alphanumeric character here, bounded by single quotes.

number_of_bytes is the number of bytes you wish to initialze with the value. This can range from 1 to 65535, but must be an actual value, rather than a variable.

Sprite Collisions with boxcollision

The Atari 7800 doesn't have hardware collision registers, so any collision detection needs to happen through software.

7800basic provides a boxcollision check that you can use to see if two of your sprites overlap. You must specify the sprites coordinates, and their width and height information...

```
if boxcollision(sprite1x, sprite1y, sprite1w, sprite1h, sprite2x, sprite2y,
sprite2w, sprite2h) then ...
```

The width and height information must be numerical values. (not variables)

Normally 7800basic adds a bit of extra code to collision checking that allows it to check for sprites that are partially off screen. If you need a performance boost, you can turn collision wrapping off...

```
set collisionwrap off
```

Be advised that turning collision-wrapping off may cause collision checks near the X 0 coordinate to also fail, so it’s advised not to turn off collision wrapping if you need collision detection in the left-most coordinates.

Boxcollision may fail if your width values much larger than typical sprite widths, due to 6502 math overflow.

Checking the overlap of too many sprites with boxcollision() may cause your game to use too many cycles, which will result in slowdown and graphic artifacts. Checking if main character collided with 10 enemies shouldn't be an issue, but checking if those enemies have collided with each other will likely be a problem.

One solution for having too many objects to detect collisions between is to limit the sprite locations to certain areas of the screen, and only collision detect for sprites in the same area.

There are faster alternatives to boxcollision as well. If you just want to see if a bullet or small object

has hit an enemy, you can do a quick coordinate check...

```
if bulletx>enemyx && bulletx<(enemyx+16) && bullety>enemyy &&
bullety<(enemyy+16) then ...
```

Another thing to keep is mind is that boxcollision() isn't reliable with off-screen objects. Avoid checking objects with coordinates that are larger than the screen boundaries.

Joystick Controls

Two-button joystick controls are the default in 7800basic. If you've switched from joysticks to another controller, you may switch back with the `changecontrol` statement, specifying controller port 0 or controller port 1.

```
changecontrol 0 2buttonjoy
```

The 2buttonjoy mode will work with one button sticks, but if you prefer to force a single-button mode, you can do so with changecontrol.

```
changecontrol 0 1buttonjoy
```

Joysticks are read by using an if...then statement. There are four directional functions and two fire button functions for each joystick.

```
if joy0up then y = y - 1
if joy0down then y = y + 1
if joy0left then x = x - 1
if joy0right then x = x + 1
if joy0fire0 then goto __Purple_Monkey
if joy0fire1 then goto __Aqua_Monkey
```

These can also be inverted using the ! token. For example:

```
if !joy0up then goto __Tasty_Pilgrim
```

If a 2600 style single-button joystick is plugged in, the joystick button is read through joy0fire1.

If you wish to see if a joystick was used at all, you can check joy0any or joy1any:

```
if joy0any then goto playermoved
```

Driving Controls

Atari 2600 style driving controls can be used with 7800basic. To enable them for your game, use the `changecontrol` command in your program code, specifying controller port 0 or port 1.

```
changecontrol 0 driving
```

7800basic will then read the driving controller every frame, and adjust either the drivingposition0 (controller port 0) or drivingposition1 (controller port 1) variables.

If you wish to use the driving control like a quick-turn paddle controller, instead of slower driving-wheel control, you may wish to change the device step-per-resolution and/or use the “drivingboost” acceleration curve for the device.

```
set drivingboost on
port0resolution = 3
```

The driving wheel is an extremely low-resolution device, with 16 positions per 360 degree rotation. The above settings would make the controller useful for a paddle game.

Take note that driving controls, paddles, and the mouse x-axis all use the same variables for position, and the buttons can all be read using the regular joystick joy0fire method. This makes is easy to support multiple devices in your game code.

Paddle Controls

Atari 2600 style paddle controls can be used with 7800basic. To enable them for your game, use the `changecontrol` command in your program code, specifying controller port 0 or port 1.

```
changecontrol 0 paddle
```

7800basic will then read selected paddle controllers every frame, and adjust the paddleposition0 (paddle 0, controller port 0), paddleposition1 (paddle 1, controller port 0), paddleposition2 (paddle 0, controller port 1) or paddleposition3 (paddle 1, controller port 1) variables.

By default, only the first paddle in the controller port is read. If your game requires two paddles, you can enable both controllers in the port.

```
set paddlepair on
```

For reading the fire button of the first paddle in a pair, you can use joy0fire. (or joy1fire) To read the second paddle fire button, you’d need to check joy0right. (or joy1right)

You can also set the paddle range, which will adjust the amount of time required to read the paddle.

```
set paddlerange 160
```

To save cpu time, you may wish to reduce the paddle range and scale up returned values x2.

```
set paddlerange 80
set paddlescalex2 on
```

Take note that driving controls, paddles, and the mouse x-axis all use the same variables for position, and the buttons can all be read using the regular joystick joy0fire method. This makes is easy to support

multiple devices in your game code.

Mouse Controls

Either Atari ST or Amiga mice can be used with 7800basic. To enable them for your game, use the `changecontrol` command in your program code, specifying controller port 0 or port 1.

```
changecontrol 0 stmouse

changecontrol 0 amigamouse
```

7800basic will then read the selected mouse every frame, and adjust the mousex0 and mousey0 variables (controller port 0) or mousex1 and mousey1 variables (controller port 1).

If your game only incorporates mice for side to side movement, you may wish to save cpu time by only reading the mouse x axis.

```
set mousexonly on
```

You can also set the amount of cpu time that 7800basic will spend polling the mouse.

```
set mousetime 100
```

The easiest and least expensive source of compatible mice are PS2 mouse -> Amiga/Atari mouse adapters, used in conjunction with PS2 mice. These adapters tend to have less issues, while providing the superior resolution of a modern mouse. Legacy ST and Amiga mice may not work with all consoles, and even when they do, these are lower resolution devices.

If you wish to support legacy mice in your game, you may wish to warn your users that old mice may not be reliable on all consoles. You can also change the resolution of the device.

```
port0resolution = 2
```

Take note that driving controls, paddles, and the mouse x-axis all use the same variables for position, and the buttons can all be read using the regular joystick joy0fire method. This makes is easy to support multiple devices in your game code.

Keypad Controls

Atari 2600 keypad controls can be used with 7800basic. To enable them for your game, use the `changecontrol` command in your program code, specifying controller port 0 or port 1.

```
changecontrol 0 keypad
```

Keypad controllers are read by using an if...then statement to test for key presses. Here’s an example of testing if button “3” is being pressed.

```
if keypad0key3 then goto handlekeypress3
```

The keypad 0 buttons can be read with keypad0key0 through keypad0key9, keypad0keys (star), and keypad0keyh (hash).

The keypad 1 buttons can be read with keypad1key0 through keypad1key9, keypad1keys (star), and keypad1keyh (hash).

The keypad data is also encoded as bits in the variables `keypadmatrix0a-keypadmatrix0d` and `keypadmatrix1a – keypadmatrix1d.`

Console Switches

the reset, select, and difficulty switches

Reading the console switches is done by using an if...then statement.

`if switchreset`
True if Reset is pressed. See reboot if you would like to use switchreset to warm boot your game.

`if switchselect`
True if Select is pressed.

`if switchleftb`
True if left difficulty is set to B (amateur), false if A (pro).

`if switchrightb`
True if right difficulty is set to B (amateur), false if A (pro).

For example, these are accessed by:

```
if switchreset then goto myreset
```

These can all be inverted by the NOT (!) token:

```
if !switchreset then goto skipreset
```

pausing

7800basic has a built-in handler for the pause switch, and will automatically pause your game and silence any TIA audio if the pause switch is pressed. If you're satisfied with this behaviour, there's nothing else required from your basic code.

If you would like to run a special routine during this time, create a subroutine called pause. This is where you can play a song, grey out colors, or display a “paused” message while the console is paused.

If you wish to draw moving sprites or changing characters during your pause subroutine, you'll need to restorescreen or clearscreen at the top of your pause routine. You don't need to include a drawscreen

command though, as this is done for you when you return from the pause subroutine.

```
pause
  restorescreen
  gosub DrawPlayerTappingFoot
  return
```

7800basic provides a “pausestate” variable you can use to detect the first time through the pause routine. This variable will equal 1 on the first time running the subroutine, and it will equal 2 on subsequent runs.

This is useful if you want to display a simple “paused” message, but don't want to bother with restorescreen or clearscreen.

```
pause
    if pausestate=1 then plotchars pausedtext 2 64 5 5
    return
```

If you wish to roll your own pause handler, you can do that as well. To stop 7800basic from including the built-in handler, use the “set pauseroutine off” command.

```
set pauseroutine off
```

You can then read the pause switch state with if switchpause.

```
if switchpause then goto myownpauseroutine
```

Remember that the switch is a momentary switch, so you'll need to track one press and release for pausing, and another press and release for un-pausing.

If you only wish to momentarily disable the pause feature, you can set the variable “pausedisable=1”. To reenable the pause feature, use “pausedisable=0”.

Direct TIA Sound

TIA is the sound chip in the 7800. If you're creating a game that has no extra soundchip on the circuit board, you'll be making all of your sounds by manipulating TIA. You can do so by playing TIA data with the playsfx command, or by manipulating the TIA registers directly.

tsound

7800basic provides the tsound command for conveniently setting the TIA audio registers. It's similar in function to the sound command found in Atari BASIC.

```
tsound channel, [frequency], [waveform],[volume]
```

The channel value must be 0 or 1. You may omit any of the frequency, waveform, or volume values, if

you wish to keep a previously set value for the current channel. (though you must still include the commas)

Here's an example of using tsound to set the frequency and volume for channel 0, and omitting the waveform information.

```
tsound 0,12,,8
```

TIA sound registers

If you wish, you may drive the TIA audio by changing its registers directly, instead of using tsound.

TIA has two independent audio channels that work identically (channel 0 and channel 1). Each channel has three registers that determine the sound that will be produced (AUDV0, AUDC0, AUDF0, AUDV1, AUDC1, AUDF1).

AUDVx stands for AUDio Volume. The volume register determines the volume or amplitude of the sound that will be produced. Valid values are 0 through 15.

AUDCx stands for AUDio Control. The control register determines the waveform or tonal quality that will be produced. Valid values are 0 through 15.

AUDFx stands for AUDio Frequency. The frequency register determines the frequency or note that will be produced. Valid values are 0 through 31.

Set AUDV0 or AUDV1 to pick the volume or amplitude of the sound you want to play—0 is "off" or "mute" and 15 is the loudest.

Set AUDC0 or AUDC1 to pick the waveform you want to use. With one exception, each waveform is just a stream of 0s and 1s that are repeated endlessly in a specific pattern, causing the speaker to vibrate in that pattern and create a sound. The length of the pattern determines the primary frequency of the sound, and the complexity of the pattern determines how "pure" or "noisy" the sound is. One waveform is just a stream of 1s (or "always on"), so it sounds "silent" (because the speaker doesn't vibrate back and forth the way it does with the other waveforms)-- but you can use the volume register with this "always on" waveform to create your own waveforms.

Set AUDF0 or AUDF1 to pick the frequency or note you want to play. The TIA doesn't use the standard set of musical notes (C, D, F#, G, etc.)—instead, it uses harmonics (or really "subharmonics") of the primary frequency that's been selected with the AUDC0 or AUDC1 register. To determine what the resulting frequency will be, add 1 to the AUDF0 or AUDF1 setting and divide the primary frequency by that number—for example, AUDF0 = 3 divides the primary frequency by 4 (3 plus 1), whereas AUDF0 = 7 divides it by 8.

Unless you're using the "always on" waveform, making music with the TIA is a "set it and forget it" thing, meaning you can set AUDC0, AUDV0, and AUDF0 (or AUDC1, AUDV1, and AUDF1) and they'll keep their values, playing the same sound continuously until you change their settings to pick a

different sound. In practical terms, this means you don't need to keep setting them over and over again within a loop to keep playing the same sound; you just set them once and then you don't have to worry about them again until you're ready to play a different sound.

Direct POKEY Sound

7800basic is able to access the POKEY sound chip, if the chip is provided either on the cartridge or through add-on hardware. To enable this access, use the following line near the top of your basic program.

```
set pokeysupport on
```

You can check to see if a POKEY chip was detected in the following manner.

```
if pokeydetected then gosub playmypokeysong else gosub playmytiasong
```

psound

7800basic provides the psound command for conveniently setting the POKEY audio registers. It's similar in function to the sound command found in Atari BASIC.

```
psound channel, [frequency], [waveform , volume]
```

The channel value may range from 0 to 3. You may omit the frequency parameter. You may also omit the waveform and volume parameters, so long as you omit or include them both.

The frequency parameter may range from 0 to 255. Both the waveform and volume parameters range from 0 to 15.

Here's an example of using psound to set the frequency and volume for channel 0.

```
psound 0,200,10,15
```

Its recommended you stick to even waveform values, as odd ones only move the speaker to a certain position, and won't play a frequency. Here are some descriptions of waveform values you may find useful as a starting point...

0	pink noise, rough whooshing
2	triangle wave, bell tones
4	noise tone mix, rumbling at lower end
6	triangle wave, bell tones (repeat of 2)
8	white noise, whooshing
10	square wave, pure tones
12	sawtooth wave, buzzy if using AUDF that’s non-divisible by 3
14	square wave, pure tones (repeat of 10)

POKEY sound registers

7800basic handles all initialization when it runs through its POKEY detection routines at start up, so you don't need to do any initialization of your own.

Because POKEY is detected and may be in different locations in memory, you'll need to access the POKEY registers using indirect array access.

The details of driving POKEY through its registers is beyond the scope of this document, but here's an example of how to set the PAUDF0 register to 100, within 7800basic's auto-detection scheme.

```
pokeybase[[PAUDF0]]=100
```

TIA and POKEY Sound Effect Driver

7800basic has an advanced sound driver that greatly simplifies adding both TIA and POKEY sound effects in your game.

```
playsfx soundata [offset]
```

The optional offset parameter allows you to raise or lower the pitch of the played sound. If you have a sound which is often repeated, it's recommended that you vary its pitch a bit randomly.

```
temp7=rand&3 : rem set temp7 to a random number from 0-3
playsfx sfx_lasershot temp7
```

When you play a sound, the sound driver will automatically choose between the two available sound channels based on which channels are already being used by the driver, and if both channels are being used it will interrupt one of the playing sounds based on a priority system.

Before playing a sound with playsfx, you'll need to define the sound effect data in the expected format. The data format 7800basic works with is composed of three parts, a header, the sound data itself, and an end-of-sound marker.

The header consists of 3 bytes:

- The format version. Use “16” here for TIA, and “32” here for POKEY.
- The sound priority. The suggested usage here is to enter the number of chunks of data. This will have the effect that longer sounds will be less likely to be interrupted early on compared to short sounds. If you have a background sound that should only be played if a channel is free, use a 0 here.
- The number of frames each chunk of data represents, less one.

The sound data then consists of 3 byte chunks, each of which represents the Sound Frequency, Control/ Waveform, and Volume to play. The sound driver will continue to play the sound data until it reaches

an end of sound marker, which consists of 3 zero bytes.

Here's an example TIA sound, which simulates the sound effect when jumpman jumps a barrel in Donkey Kong. There are 5 chunks of sound data, and each chunk plays for 5 frames.

```
  data sfx_jumpman
   16,  5,  4 ; version, priority, frames per chunk
  $1E,$04,$08 ; 1st chunk of freq,channel,volume data
  $1B,$04,$08 ; 2nd chunk
  $18,$04,$08 ; 3rd chunk
  $11,$04,$08 ; 4th chunk
  $16,$04,$08 ; 5th chunk
  $00,$00,$00 ; End Of Sound marker
end
```

If you wish to constrain the sound driver to only using the first sound channel when playing TIA sounds, you can use the "`set tiasfx mono`" statement.

If you wish to silence all in-progress sounds, you can use the "`mutesfx`" command.

Music Tracker

7800basic has a music tracker you can use to automatically play song data, with options to select tempo and number of repetitions.

The music tracker requires that you provide instrument data and song data.

enabling the tracker

Since not every game design feature music, 7800basic doesn't automatically incorporate the tracker code into your game. To enable music tracker support, add the following line near the start of your basic program:

```
set trackersupport basic
```

instrument data

Instruments are defined in data statements that are similar in structure to sound effects. Here's the syntax breakdown.

byte 0: Instrument type+version. $10 for TIA is all that's supported right now)
byte 1: Instrument priority. Instruments work with the 7800basic sound-effect priority system.
byte 2: Frames per chunk. Larger values here plays back the instrument data slower.
bytes 3+4: offset+volume for the first chunk of instrument sound data. The offset is semitone offset from the note currently being played, so you can simulate tremolo.
bytes 5+6: offset+volume for the second chunk of instrument sound data.
...
bytes "0","0": The End Of Instrument flag

Here's an example instrument.

```
 data tiashort
  $10,$00,$03 ; version, priority, frames per chunk
  $00,$06 ; note offset, volume
  $00,$04 ; note offset, volume
  $00,$00 ; End Of Instrument
end
```

You may have noticed that there isn't a place to select which TIA channel/distortion is being played. Because TIA doesn't have much useful overlap between the frequencies it produces, the TIA channel is automatically selected for you based on the note position being played.

songdata

Song data is stored with your basic source code with a songdata command, which follows the same basic syntax as other 7800basic data statements.

```
  songdata mysong
  [song data here]
end
```

The actual song data has it's own special syntax, which will be covered in following sections in-detail. It should be noted that addional white space within songdata is ignored, so you are free to data together using spaces, tabs, and newlines.

labels

Any word in the song data that isn't indented from the left-margin is considered a label.

There are 4 special labels that identify the data the song should begin with – main1, main2, main3, and main4. The 4 labels are played in parallel, as 4 independant tracks.

Even though there are 4 tracks available, you still need to keep in mind there are only 2 TIA voices available. Having 4 tracks will become more useful in the future when other instrument types, like POKEY, are integrated into the tracker.

The song data that follows a label may contain notes, note modifiers, or other labels. Here's an example of a songdata statement with 2 tracks, and labels that call other labels.

```
    songdata funeralmarch
main1
  k=a4
  i=tiashort
  c2 c8 r4 c8 c2   d#8 r4 d8 d8  r4 c8 c8 r8 < b4 > c2
main2
  i=tiashort
  k=a2 bassline ; transpose the key, and call the baseline label
  k=b2 bassline ; again, with another transpose
  k=a2 altbassline ; and again
```

```
  c2 ; final note
bassline
  c4 c4 c4 r4
altbassline
  c4 c4 < b4 > r4
end
```

If you want a called label to play a number of times, by adding a period and number after the label name. This method uses less ROM space than repeating the same label call over and over, but is limited to playing the label up to 8 times.

Here's an example of a labels called multiple times.

```
    songdata herecomesthesun
main1
  k=a3
  i=tiashort
  intro.4
  chorus.8
intro
  [note data]
chorus
  [more note data]
end
```

It should be noted that labels can only call other labels to a depth of 3; any one of the main tracks (depth 1) can call another label (depth 2), which in turn can call another label (depth 3). If you go beyond a depth of 3, the song and your program itself will likely crash.

track and song modifiers

At any time in the song, you may set/change the instrument being played on the track, transpose the key a track is in, or modify the overall song tempo.

To change the instrument being played, set "i" equal to the name of the data containing the instrument definition.

```
i=tiashort
```

To transpose the absolute key the track is being played in, set "k" equal to the note position on the TIA scale.

```
k=a3
```

There are a few ways to change the relative key of the track. Here's are examples of shifting the key up 2 semitones, down 3 semitones, shifting the key down an octave, and shifting the key up an octave.

```
k+2
k-3
<
>
```

To transpose the absolute tempo the track is being played in, set "t" equal to the BPM value you desire.

```
t=120
```

You can also change the relative tempo of a track. Bear in mind the numbers in a relative tempo adjustment don't represent BPM, but rather just internal tempo counter amounts.

```
t+5
t-10
```

TIA key mapping

TIA isn't capable of playing pure tones across the whole musical spectrum. Some musical frequencies aren't able to be played at all, while others must be played in a TIA channel with distortion or buzzing. The following is a breakdown of the notes the 7800basic tracker can play, and their tonal quality.

```
Octave 0 – Electronic
note is very out of tune:      . .  . .  . * .  . .  * .  .
                               c c# d d# e f f# g g# a a# b

Octave 1 – Somewhat Buzzy
note is very out of tune:      . .  . .  . . .  . .  . .  .
                               c c# d d# e f f# g g# a a# b

Octave 2 – Buzzy
note is very out of tune:      . .  . .  . . .  . .  . .  .
                               c c# d d# e f f# g g# a a# b

Octave 3 – Buzzy from c3 to d#3, Pure from e3 to b3
note is very out of tune:      . .  . *  . . .  . .  . .  .
                               c c# d d# e f f# g g# a a# b

Octave 4 – Pure
note is very out of tune:      . .  . .  . . .  * .  . .  .
                               c c# d d# e f f# g g# a a# b

Octave 5 – Pure
note is very out of tune:      . .  . .  . . .  . .  . .  .
                               c c# d d# e f f# g g# a a# b

Octave 6 – Tuned Wind
note is very out of tune:      . .  . .  . . .  . .  . .  .
                               c c# d d# e f f# g g# a a# b

Octave 11 – Drum Kit
note drum pattern number:      0 1  2 3  4 5 6  7
                               c c# d d# e f f# g

      0=Bass/Kick, 1=Low Tom, 2=High Tom, 3=Snare,
      4=Closed Hat, 5=Open Hat, 6=Bass/Kick+ClosedHat,
      7=Snare+Closed Hat
```

note data

Notes are represented with regular music lettering and using the # sign to indicate sharps. The number following the note indicates the length, where 1=whole note, 2=half note, 4=quarter note, 8=eighth note, and 16=sixteenth note. At preset time, other fractional notes – third, fifth, etc. - aren't supported.

Here's an example of how to play an octave scale with quarter notes.

```
c4 c#4 d4 d#4 e4 f4 f#4 g4 g#4 a4 a#4 b4
```

To play the notes below c, you need to temporarily shift down an octave.

```
c4 < e4 g4 > c4
```

To play rests, you just use the r letter as you would a note.

```
c4 r4 e4 g4
```

drum note data

An alternate note representation can be used for drum beats. Periods indicate rests, and numbers indicate the drum instruments. The default drum instruments are provided at the key of c11.

```
  songdata drums
main1
  k=c11
  '6.4.6.4.3.4.6.6.6.6.6'
end
```

playsong

Once you've put together the song and instrument data, you can play the music using the playsong command. When doing so you must supply the initial tempo in BPM.

```
playsong songname 120
```

You can also tell the command to repeat the song indefinitely.

```
playsong songname 120 repeat
```

Or you may tell it to repeat a limited number of times.

```
playsong songname 120 4
```

stopsong

If you wish to halt a playing song, you can do so by using the stopsong command.

```
stopsong
```

AtariVox Voice Support

You can send speech data to the AtariVox via the speak command.

```
speak speechdataname
```

See the *speechdata command* for more information.

High Score Support

7800basic supports high score tables for up to 4 different difficulty levels in your game. The actual device used to store the tables can be an Atari 7800 High Score Cart, AtariVox, or SaveKey device.

To incorporate high score support in your game, you first need to enable high score support. Pick a unique 4 digit hex value to identify your game...

```
set hssupport $1234
```

When your game is nearing completion, you can reserve your chosen number by announcing it in this AtariAge thread: http://atariage.com/forums/topic/128432-high-score-cart-values/

You also need to include the high score font graphic in your game. It can go in any graphics block, but **must** be the first graphic imported in its block. Note that you need to use 320A mode for the high score graphic, even if your game uses a different mode.

```
incgraphic hiscorefont.png 320A
```

Before trying to display a high score table, you need to set the game difficulty level variable from 0 to 3, which represents easy, intermediate, advanced, and expert levels.

```
gamedifficulty=2
```

If your game has no difficulty levels, you can leave gamedifficulty at the default of 0.

You can then display the current high score table, without allowing high score entry with the command...

```
drawhiscore attract
```

When a single player game is over, and you them to have the chance to enter their initials, you can use the command...

```
drawhiscore single
```

Or with a two player game, you can use the following command to allow the first player to enter initials...

```
drawhiscore player1
```

And as you might expect, the second player has a similar command...

```
drawhiscore player2
```

The drawhiscore command will erase the currently drawn screen, savescreen buffer, and the color palettes, so you'll need to draw a new screen after issuing this command.

It's also noteworthy that drawhiscore command allows the high score display to be interrpted by pressing the joystick fire button. The command may exit with the user holding the fire button down, so you may need to account for that in your basic code.

If you want the players' scores to be provided a descriptive ranking, you can use the “set hsgameranks” command. Scores should be ordered from biggest to 0...

```
set hsgameranks 30000 commander 10000 lieutenant 0 ensign
```

If the player's score variable didn't beat any recorded scores, the high score routines will just diplay the table, and the ranking description if you've enabled that functionality in your game.

If you have AtariVox speechdata named “vox_highscore”, the AtariVox will say the speech once while the player enters a high score.

If you have sound effect data named “sfx_highscore”, the sound effect will play once while the player enters a high score.

If you have a tracker song named “song_highscore”, the song will play once while the player enters a high score.

There are various “set” commands that change the behavior of the high score tables. See the *Program Configuration* section for more details

Lastly, if you wish to detect which device is being used, you may check the `hsdevice` variable:

0=no device, 1=High Score Cart, and 2=AtariVox/SaveKey

Saving Game Memory

7800basic supports saving and loading up to 25 variables to any detected High Score device. You will need to include high score support in your game, as demonstrated in the previous *High Score Support* section. If you only wish to save game memory and not display high score tables, you may omit importing the high score font.

The data will be saved in an unused difficulty level's area. To inform 7800basic of this, we set the gamedifficulty variable.

```
gamedifficulty=3
```

From there we call the savememory command to save your variables.

```
savememory playerx playery playerroom playerworld
```

If the variables are dim'ed consecutively in memory, you may list them as a range.

```
savememory playerx-playerworld
```

Loading previously save variables is done with the loadmemory command, which also accepts both forms of variable listing that the savememory command uses.

```
loadmemory playerx playery playerroom playerworld

loadmemory playerx-playerworld
```

If your game is also displaying high score tables, be sure to restore the `gamedifficulty` variable to its original value after loading/saving data.

Feature Modules

7800basic has a lot of options to interface with external hardware, and there isn't enough space in the 7800basic's reserved 4k area to hold all of the feature modules required. If get a message about this area having negative space during compiling, it means you need to store the modules elsewhere in your game.

You can do so with the "inline" keyword, at the location where the module can go. At present time you can do this for the atarivox, pokey support, and sound tracker modules.

```
inline 7800vox.asm
inline pokeysound.asm
inline tracker.asm
```

Avoid putting these statements directly in the path of your own code flow, since they're not meant to be executed directly.

Conditional Compilation and Compiling Message Output

7800basic has a few commands so you can easily enable or disable code blocks from being added to the ROM. This allows you to build a few different custom rom image, from just a minor change to a constant value in your source code.

These commands are #ifconst, #else, and #endif, and can be used in the following fashion.

```
const officialrelease = 1

[a bunch of common game code would go here]

#ifconst officialrelease
  plotbanner officialbanner 0 0 0
#else
  plotbanner demobanner 0 0 0
#endif
```

There's also an `echo` command which you can use to display messages, or the values of constants, when the ROM is compiled.

```
 echo " "
 const airblockstart    = <chars_air
```

```
const airblockend      = airblockstart+chars_air_width-1
echo "TILES: airblockstart:",airblockstart,"   airblockend:",airblockend
echo " "
```

The `#ifconst`, `#else`, `#endif`, and `echo` commands are all passed directly from your 7800basic source code to the dasm assember, which is responsible for their function. If you wish to learn more about them, you can reference the documentation for dasm.

Assembly Language

7800basic allows you to incorporate your own assembly language routines into your basic code in a few different ways.

inline assembly

The first is to just use the asm command, followed by assembly language, finally ending with the end keyword.

```
  drawscreen
  asm
  lda PlayerX
  lsr
  lsr
  lsr
  sta PlayerCharX
end
```

including external assembly

The second method, the include command, allows you to keep the assembly language in a separate file. The assembly code will be imported to an area outside your basic code, and you'll be expected to call a label in the assembly with either a function or gosub call.

```
include killallhumans.asm
```

The include command must typically precede all other commands (at the beginning of your program, before anything else). At this time there is no checking to ensure that you do this. Particularly, if you use an includes file or a different kernel, you need to specify all of your include commands first or they will be ignored.

inlining external assembly

The inline command is like the include command, except 7800 import the assembly code directly into the program where the inline command is.

```
inline savehumanity.asm
```

Installing 7800basic

7800basic is distributed as a single zip file. Download the latest zip file and unzip to whichever location you desire to use. Make sure your unzip utility creates the expected subdirectories (/docs, /includes, ...) rather than sticking all of the files into one directory.

Windows users should double-click and the provided install_win.bat file. OS X and Linux users should run the install_ux.sh script.

In both cases, just follow any instructions presented by the installer.

Compiling Your 7800basic Code

7800basic is a command-line program, so these instructions will cover how to compile your basic programs using the command-line. If you're uncomfortable with basic command-line usage for your OS, you may wish to read up on it a bit first.

on Windows

Assuming you've completed the 7800basic installations in the previous section, you can compile your program under Windows with the following steps...

1. click on `Start->Search` or `Start->Run` and type: CMD, followed by [ENTER]
2. type: cd \my\project\directory (substitute the directory that contains your BASIC source)
3. type: 7800bas.bat myprogram.bas (substitute the filename containing your BASIC source code)

Information about the compiling process will be displayed in your CMD window. With some luck, you should have some new compiled files.

on OS X or Linux

Assuming you've completed the 7800basic installations in the previous section, you can compile your program under OS X or Linux with the following steps...

1. open a terminal or console window.
2. type: cd /my/project/directory (substitute the directory that contains your BASIC source code)
3. type: 7800basic.sh myprogram.bas (substitute the filename containing your BASIC source code)

Information about the compiling process will be displayed in your terminal window. With some luck, you should have some new compiled files.

files produced by compilation

myprogram.bas.bin

this is a ROM file for your game, typically used for creating cartridges.

myprogram.bas.a78

this is another ROM file for your game, typically used for emulation. This format file has a special header, to let emulators know which features are needed for the ROM, like controllers, bankswitching format, etc.

myprogram.bas.list.txt

this holds the detailed assembly listing of your game. This file can be potentially useful in hunting down syntax errors and similar bugs.

other temporary files

There are other temporary files are created by the compilation process, but these aren't particularly useful to the 7800basic coder.

Troubleshooting Issues With Your 7800basic Code

Here are some issues that you may encounter while using 7800basic, and what their underlying cause may be.

- The game screen turns red at some point, and the action freezes up.

  You've somehow managed to overwrite memory that 7800basic owns by growing the 6502 stack excessively. This can happen if you have a large number of nested subroutines, or if you haven't used "return" after a previous "gosub". It's possible to turn off this protection with the "set canary off" statement, but you should figure out the reason the protection was tripped in the first place.

- The game runs slow, movement stutters, and/or some of the sprites seem to flicker.

  You've run out of time between your "drawscreen" statements, so some remaining plot* statements weren't able to run until the next frame. This causes an extra frame to be inserted when the remaining plot* statements do run, except none of the game logic was run during that frame.

  If you have too many sprites, then try reducing the number of them in your game. Otherwise ensure that you do as much game logic as possible prior to any plot* statements, and keep the

logic around your plot* statements as lean and efficient as possible.