7800basic Guide

based on 7800basic v0.2 beta

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.

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.

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 2-byte memory variable. 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.

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.

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 assign the value of rand to something else, 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" it such that 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
```

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.

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
```

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 to which letters you wish to use to represent these graphics with the alphachars command.

```
alphachars ' abcd'
```

and then you could create text data with those characters with the alphadata command.

```
 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
```

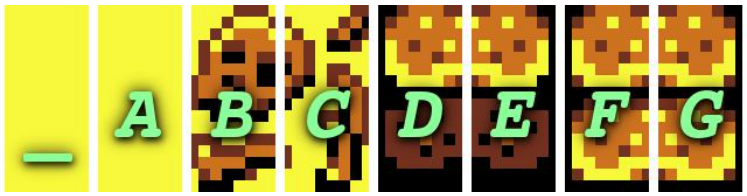
Alphadata can also be used to help entry of non-alphabet data.

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'
```

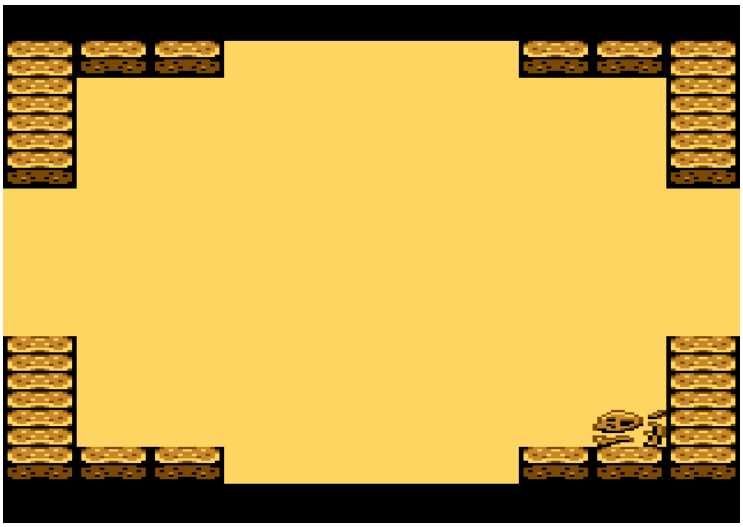
Which would tell 7800basic to reference tiles in the graphic with the following letters...



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 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”.

Program Configuration

7800basic has a series of “set” commands you may use to customize the 7800basic environment configuration. These should be called close to the top of your program, and each particular set command should only be called once.

set romsize [value]

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

```
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.

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

The default zone height is 16, but can be defined just 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 pauseroutine [on|off]

This tells 7800basic to not use its built-in pause routine in the game.

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 tiasfx mono

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

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.

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
```

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.

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 index #0] [#1] [#2] ...
```

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 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
```

plotsprite

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

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

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.

The display order of displayed 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.

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 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

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.

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.

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.

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.

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)

If your game needs collision detection with sprites that are partially on-screen partially off-screen, you'll need to set collision-wrapping on.

```
set collisionwrap on
```

Be advised that setting collision-wrapping on makes the collision detection take a bit of extra time.

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 ...
```

Joystick Controls

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.

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.

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.

playsfx

7800basic has an advanced sound driver that greatly simplifies adding TIA 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 now.
- 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.
- 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 TIA Frequency, Channel, 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 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
```

Lastly, if you wish to constrain the sound driver to only using the first sound channel, you can use the “set tiasound mono” statement.

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.

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.

```
pokebase[[PAUDF0]]=100
```

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.