Whether you're young, old, experienced, or a novice, the new Wildbits/K2 will entertain and delight you and your audience of family and friends.

At first glance, you'll be drawn to the K2's refined look and colorful exterior. Look closer, and you'll find that this home computer is different from the rest.

Getting Started Volume I along with its linked content begins by highlighting a subset of its many capabilities.

Across the next several pages, we will cover the basics of navigation and operation through simple examples and links to YouTube-based content.

We will then move on to a few type-in programs; short SuperBASIC demos to modify and learn from.

The back pages of this guide provide links and QR codes to Internet based resources across social media, the Wildbits wiki, and GitHub. There is a growing body of work which was contributed to by a community of Wildbits users; a community that you are now officially a member of!!

This guide organizes its content loosely in 3 parts:

I. Introduction to the system (summary of capabilities)
II. Power-on and navigation (including sample programs that you can type in and run)
III. Single-screen SuperBASIC programs (longer demos which are suited for modification)

Part I - Introducing the Wildbits/K2 system

Informed by some of most capable home computers in history, your K2 has a heart that beats - a real silicon based CPU driven by a high-speed system clock (up to 12.6 megahertz, depending on *core*[1]).

The K2's CPU is a genuine Western Design Center (WDC) 65C816 CPU, a successor to the MOS Technology 6502 microprocessor; the most popular 8-bit home computer CPU in history.

MOS was purchased by Commodore in 1976 and went on to release numerous support ICs in addition to the VIC-II video chip and the 6526 Sound Interface Device (SID). We will talk about these later!

If you are a retro enthusiast, you might know that the 6502 made its mark powering the original Apple and Apple 2 Computers, Commodore machines from the KIM-1 to the original PET, VIC-20 and Commodore 64 and others, and ATARI 8-bit systems including the original 2600/VCS. The 6502 also drove the beloved BBC Micro and was responsible for some of the most memorable coin arcade titles, such as Asteroids, Centipedes, and Missile Command.

Your K2 is powered by the square packaged variant of the original 40-pin 65816 used in the Apple IIGS and in the Super Nintendo Entertainment System, but similarities to these greats do not end there.

[1] In addition to the main CPU, your K2 has an *FPGA*; it can be thought of as a super-CPU which enables a software defined hardware capability that is field-programmable (it can be upgraded with new features). Its code definition is known as a 'core'.

Graphics, Sound and more…

Within a layered graphics system, the K2 hosts planar based, high-color **bitmapped** and **tile** engine graphics (inspired by Nintendo and other late ‘80s and early ‘90s gaming consoles). The K2 also supports 4 **sprite** layers capable of displaying as many as 128* full-color, independently movable objects (16 times that of the venerable Commodore 64). You’ll also find redefinable character graphics and flexible video modes on a topmost layer that defaults at 80 x 60 characters but can be configured with large double-wide/double-high characters, useful for games or for visually impaired individuals.

From an audio and sound perspective, the K2 orchestrates a symphony that surpasses the music generation capabilities of the Apple IIGS, the Amiga, and wavetable based PC sound cards, and can do so simultaneously. It can even play MP3 encoded audio thanks to onboard hardware decoders.

And the fun does not end there; there is a pair of **MIDI** DIN ports, a high-resolution color 280 x 240 mini **LED screen** integrated into the case, and **wireless networking** with an external antenna. There are 9-pin connections for serial devices, ATARI style joystick ports, and amplified headphone or line-level RCA stereo output jacks.

You can program your K2 in its native environment, SuperBASIC; an interpreter inspired by BBC BASIC, offering procedural capabilities and an inline assembler. We will explore SuperBASIC in depth, on the pages below.

For more ambitious development, full-featured applications can be created using powerful open source **cross-assemblers** such as 64tass. You can also develop code on your modern computer in the **C Language**, and a growing list of other languages.

There are dozens of community members discussing and sharing work on GitHub, and discussions about all of this activity on on a dedicated Discord server, the Wildbits community home.

Did you know that your K2 is a chameleon?

The K2 runs standard 6502 and 65C02 machine code but can host 3 additional environments (called 'contexts') with alternate cores and operating system builds, selectable at boot time. As shipped, the K2 leverages a **65816 model with a flat 24-bit memory map** (shipped in the 2nd slot on your system). In addition, the FPGA based 6809 **NitrOS-9** operating system can be added at no additional cost.

With all of this flexibility, the K2 is an ideal porting target and there are already K2 native versions of some popular 1980s titles such as Pitfall, Lode Runner, Ultima III, z-machine based text adventures, public domain Apple II software, and more to come.

And most fun of all, vintage computer magazines and development journals such as Compute Magazine and Dr. Dobbs Journal contain 6502 and 6809 example code ripe for porting and experimentation. You are unlikely to run out of ideas or potential.

* number of sprites is core dependent, but at a minimum, 64 are supported even before resorting to raster tricks which can really push the limit

Ports, connections, and additional information

Let's briefly examine the ports, connections, and case features and then move along to power-up your new machine for the first time. Now is a good time to grab the *quick reference/DOS reference card* that shipped with your system.

As we prepare for the next steps, you'll need the supplied power adapter and your own monitor and video cable. You can probably get by with a set of 1/8" (3.5mm) wired headphones, but eventually will want to wire the K2 to stereo powered speakers.

A pair of ATARI style joysticks is also a good bet for most games, however, you'll see that there are additional (optional) input devices supported such as a PS/2 mouse and accessories that support DPAD style NES and SNES controllers.

Consult the quick reference card for additional information including a map of ports and connections, for wireless configuration detail, and for a description and command list for the "`/ -`" launch facility and the "`/DOS`" utility.

Please NOTE: The SuperBASIC portion of this Getting Started guide includes clickable links to YouTube videos and additional content. Some of this material was produced as early as 2022-2023 and references predecessor systems from the 'Foenix Retro' project, including the F256 platforms.

Most of the material is 100% applicable to your K2 system, but if you have questions and need support, the best place for help is the issue-solving channel on the Wildbits Discord.

Part II - First power-up & an intro to SuperBASIC

Once you've confirmed power and monitor hookup is ready, it's time to turn on your Wildbits/K2.

On the rear panel (right hand side of the machine), you will find the power switch rocker. Turn it on! You should see a welcome screen similar to the following:

Notice the block cursor in the leftmost column, just below the welcome banner. The cursor is your portal and entry point to communicate with the K2 command interpreter. If you did not know it already, the SuperBASIC language interpreter runs the show, evaluating each line upon your pressing RETURN.

On the next page, you'll find a few quick keywords to get started, but first, a quick word about the K2's BASIC version, **SuperBASIC**.

SuperBASIC is an evolution of the interpreter that shipped on the famous Acorn BBS Micro in 1981.

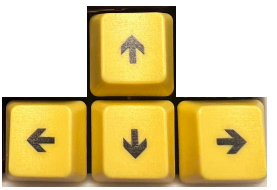
The Wildbits version is deemed '**Super**' because of system extensions which support proprietary features such as sound, graphics, and more. This introductory 6-minute YouTube video will give you a cursory overview of SuperBASIC capabilities.

The SuperBASIC environment is simultaneously an interactive workspace where keywords can be entered to perform scratch calculations or examine files present on storage devices, and, a screen based line editor where numbered lines may be entered, edited, and listed as part of a BASIC or hybrid BASIC/assembly language program.

The manner in which screen navigation functions is similar to early commercial home computers, namely, the ATARI 400/800 and the Commodore VIC-20 and 64. You'll note, however, that on the K2, the amount of screen real estate is much larger, so it is possible to enter and view an entire 80 character line of BASIC text from left to right. It is also possible to squeeze a fair sized BASIC program onto a single screen, without the need to scroll, as you will see.

Screen Navigation basics

Your K2 keyboard has 'inverted-T' cursor keys, but also, dedicated keys for navigating to the home or end of a line. Positionally, the



backspace key (labeled **DEL**) will remove the character behind the cursor and CTRL-D will delete characters from the right.

INST/DEL is located in the upper right-hand region of the keyboard, one key to the left of the F1/F2 key. Visually, this may appear unusual, but once you start using your K2, you'll quickly find that this key placement is one row up and two keys to the right of the ANSI (PC keyboard) right square bracket key '**]**', as usual. Said another way, it is exactly where you would find it, if typing by feel.

You also have access to several special keys across the 67-key map; some are implemented for specific use and others are labeled, but depending on the version of code in your machine, may or may not be bound to a specific function.

This navigation-centric video shares additional information relating to control keys, editing, and more. It also provides an introduction to the SuperBASIC keywords (there are over 100 of them).

Storage Device basics

SuperBASIC supports devices through a handful of special keywords, however, the first three (in the list below) are most often used.

Device numbering is explained in the video linked below and the `drive {#}` keyword is used to change from the default ('`0`' is the built in SD card).

• List the contents of a disk directory

```
dir
```

• Load a SuperBASIC program

```
load "{filename}"
```

• Save a SuperBASIC program

```
save "{filename}"
```

- Load a binary file to a specific memory address

```
bload "{filename}", {address}
```

- Save a binary file starting from a given memory address, for a given number of bytes

```
bsave "{filename}", {address}, {bytes}
```

This video introduces disk concepts and keywords, including how your K2 manages device numbers across the IEC[1] bus, should you have the need.

How to start, break, or list SuperBASIC programs

Assuming you've loaded a program into memory, or even if you've created your own "hello world" type of program, you will eventually want to run it.

As with other BASIC versions:

`run` <press RETURN>

To stop a running program, you can either press CTRL-C, or press the RUN/STOP key. You will see a message indicating the line number where the program execution has halted.

Listing a program is accomplished by typing the keyword `list` which supports several different forms starting with the `{from}, {to}` form.

`list 20,100` <press RETURN>

[1] The IEC port is a Commodore compatible 6-pin peripheral interface. Your K2 will work with compatible devices including modern, ESPRESSIF-based devices such as the MEATLOAF wireless project. This video discusses MEATLOAF integration.

This will list program from lines 20 through 100, inclusive. Note that the comma is used between values.

`list ,200` <press RETURN>

↑ this space is required

This form lists program lines from the beginning of the program, up to and including line 200.

Reminder: press CTRL-C or RUN/STOP to stop a program listing in progress.

`list 120,` <press RETURN>

This will list line numbers beginning with line 120 through the end of the program.

You may also use the `list` keyword along with a procedure name, as demonstrated in this somewhat advanced video. We will cover this later, as well.

Writing the inevitable two-line program

Depending on whether you just powered on your machine or have entered a few numbered lines of BASIC, now would be a good time to type the SuperBASIC `new` keyword. This will clear out BASIC memory, resetting variables and erasing references to code that may be occupying SuperBASIC's memory space.

If SuperBASIC detects that you have unsaved work in memory, it will prompt you before continuing. This will give you an opportunity to save your work.

If you've loaded binary data into the upper regions of the first 64K of memory, it is likely that it will still be there, even after a `new` keyword is issued. This is an

advanced topic, but it is possible to load graphic, audio, or other data into the memory that is shared with SuperBASIC, without its knowledge.

For more on the topic of memory management, have a watch of this video. Again, this is an advanced topic, but one that you might find value in exploring, even if you are new to the K2 platform.

Otherwise, here we go (you knew it was coming):

```
10 print "Hello Wildbits!"
20 goto 10
```

If you type `run` and press RETURN, you'll encounter an endless loop and will need to either type CTRL-C or press RUN/STOP to break.

Enhancing `print` with simple formatting

Let's make two quick modifications to our two-line program. First we will use the comma character outside of the quotes. In this usage, each iteration of `print` following the first, will begin at the next 8th character tab stop (discussed in the video linked on pg. 9 above).

Your code should look like:

```
10 print "Hello Wildbits!",
20 goto 10
```

After trying it, modify line 10 by replacing the comma with a semi-colon (note the whitespace and hyphen between the exclamation point and the closing quote; for formatting purposes, this is important):

```
10 print "Hello Wildbits! - ";
20 goto 10
```

The semi-colon keeps the ‘cursor’ (that being, the logical position where the next character will be printed) on the line following the closing quote. Normally, each new `print` statement causes a newline (sometimes known as a carriage return) to be printed. *The semi-colon suspends this behavior*, allowing you to combine values and literal text on the same screen line. Note that the text will eventually ‘wrap’ to the next screen line, and if printed at the bottom of the screen, will cause the screen to scroll. We will see more of this behavior, below.

Printing graphic characters

Your Wildbits computer uses an enhanced version of the ASCII character set and it includes dozens of graphic characters or ‘glyphs’. Each is unique and can be referenced by number ranging from 1 to 255. The BASIC keyword `chr$(x)` allows you to print the otherwise un-typeable characters by referring to them by value. In the next section, we will discuss characters in this set and how they can be used. Here is a quick example of some:

If you’ve used a vintage home computer, you may know that some had graphic characters in the extended range (128..255) on the keyboard keycaps. And Commodore supported embedding, where a given shift sequence let the user type graphic characters directly onto the screen and within quoted `print` statements.

card suit chars

The K2 is similar, except the symbols are *not* present on keycaps. Regardless, the set is expansive, with twice as many (255) glyphs across the range and an additional, secondary set of another 255.

The K2 also has a few dozen 'control codes' that, when printed with the standard `print` keyword, will cause the cursor to advance in a given direction, perform an editing functions (such as to clear to end-of-line or clear the entire screen), or change the foreground or background color. This can be used for highlighting or to control cursor location. You can also use control codes to create simple, key-based animation sequences (we will examine this below).

The entire range of Wildbits **printable** characters can be printed by using the `cprint` keyword (short for 'character' print). The following single-line program will print the K2's full primary character set across 4 screen lines:

```
10 for x = 1 to 255:cprint chr$(x);:next
```

While this *is* a complete program, it can also be executed in immediate mode (without a line number or the requirement to start it with the `run` keyword), if you desire.

Try it yourself by omitting the line number and instead, just type the colon separated keywords and then press the RETURN key.

You should see something similar to the following:

In examining the character set, you might notice some patterns:

- logical groupings of horizontal and vertical lines
- ‘shaded’ or dithered checkerboard patterns
- quarter block characters
- various arrows and bullet shaped glyphs

Below, we will make this output more useful and later, we will redefine a few for a quick diversion.

Printing characters with their ASCII values

Let’s prefix the Wildbits glyphs with numeric values and improve the formatting. Enter and run the following, but mind the spacing on line 20; it matters:

```
10 for x = 1 to 255
20   cprint x": " chr(x),;
30 next
```

The screen output should look like the following:

“10 PRINT” - a famous Commodore maze

If you are not familiar with “10 PRINT”, a quick search across the Internet will point you to some interesting reading material. A lot has been written about it, including this book and a few YouTube adaptations. Some say that “10 PRINT” is the `"hello world"` of single-line BASIC programs.

Developed in Microsoft BASIC, it can be adapted to your K2 system relatively easily, and you can choose your own characters to obtain interesting results. (by default, it uses forward and backward diagonal lines, just like the original).

The program does its thing within an endless loop that randomly chooses between just two values. Each iteration ends with a trailing `';'` as we discussed above. The result is interesting as you will see.

Three changes were necessary to get it working on the K2 platform:

- replace `PRINT` with `cprint`. Recall that the K2 needs this to print character graphics.
- choose a suitable character value (replacing the 205.5). A value of 186.5 is between “`/`” and “`\`”
- refine the calculation to accommodate a SuperBASIC floating point peculiarity.

Here is the original for Commodore platforms:

```
10 PRINT CHR$(205.5+RND(1)); : GOTO 10
```

And here is our Wildbits version:

```
10 ch=int(186.5+rnd(1)):cprint chr$(ch); : goto 10
```

Try experimenting with different values chosen from the chart on pg. 15 and examine the results.

Widening the possibilities with another example

Now let's modify the calculation to *select 1 of 5 chars.* We'll use `random(5)` to allow a wider range of values; this will produce an interesting woven pattern.

Try it yourself:

```
10 ch=int(164+random(5)):cprint chr$(ch); : goto 10
```

See the difference? In the original example, `rnd(1)` yields a number between 0.0 and 0.99999. When this is added to 186.5, it equates to 186.5 to 187.4 (which translates to ASCII char 186 or 187).

In the 'choice of 5' example, we begin with a base of 164 (a thick left) and then add whole values from 0 to 4. This selects ASCII values between 164 and 168. SuperBASIC's `random` keyword simplifies matters.

Part III - Single screen BASIC programs

A single screen BASIC program is a one that when listed, will occupy no more than one screen vertically, including all code (but perhaps, not data).

These longer examples are similar to ‘type-ins’ that first appeared in late ‘70s home computer magazines such as COMPUTE!, ANTIC, The Rainbow, or Nibble.

The attraction of single-screen programs, aside from a limited amount of lines to type in, is the level of clarity gained from being able to visually inspect variable definitions, subroutines, and core code without worrying about code scrolling off the screen.

A great feature of your K2 system is its generous 80 column by 60 row screen, the equivalent of 100-120 lines of densely packed BASIC (in the old days).

In this section we will present three increasingly complex examples and related exercises:

1. “Card” - this 24 line stepping stone program uses a combination of control characters and printed character graphics to render a single playing card on the screen. `for`/`next` loops and procedure definitions (`proc`/`endproc`) are introduced.
2. “Invaded” - this 39 line example introduces redefined characters by transforming six rarely used characters into a familiar foe.

   After loading the image data into character font memory, the screen is cleared and an alien is drawn on the top screen row. It then moves from left to right across the screen, with accompanying sound effects.

The `sound` keyword creates the familiar atonal beeping at each step. Calculations within a gating `for`/`next` loop increase the rate of speed.

Invaded addresses the memory mapped character screen using `peek` and `poke` keywords. These are two of the most powerful in any BASIC language.

3. "Balloon gone WILD" - uses Bitmap and Sprite Graphics along with new SuperBASIC keywords to mobilize a hot air balloon on-screen. Sophisticated visuals appear thanks to some color palette trickery. There are two versions: a short, single screen version which uses `goto` keywords, and a full featured `proc`-based version (full listings below).

   The accompanying sprite discussion is lengthy and covers a great deal of related topics, but it is worth a read-through, even for advanced users.

About *hexadecimal* notation (used in example 2 and 3)

Computers are most easily dealt with when spoken to using binary notation. For expediency, we meet them in the middle and use hexadecimal notation.

Hexadecimal numbers begin with a dollar sign and commonly followed by 2 (byte) or 4 (a 16-bit address) digits. When working with graphic assets, it is desirable (and required) to place data into high memory. In these cases, 6 digits are used (expressed as: `$03:0000` or "`$30000`"). Digits range from the number `0` to the letter '`f`'. So be aware, when you see a dollar sign, expect values that include:

`0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f`

otherwise known as 10 → 11, 12, 13, 14, 15

Example #1 - Card (Using character graphics)

If you had the need to print a playing card on the screen, perhaps for a simple card game such as WAR or BLACK JACK, how would you go about it?

It would seem to be a complex problem. But broken into steps, the solution is manageable. This holds true for every problem. The solution is to break it down into more simple and value-bound steps.

There are a few things to mind, starting with selecting several graphic characters (from our example on pg. 15), and keeping them straight.

Another is the requirement to print characters in sequence, such as the top or bottom edge of a card, and each left and right edge. An understanding of how `print` and `cprint` work is also important. This is something you will learn as you experiment.

Finally, there is the matter of populating the face of the card with its suit and value. For this example, we will simply display the Ace of Hearts, but if building out a full card game, you would want to substitute variables and perhaps, store information relating to placement of card notation in `data`* statements.

Once you've mastered an understanding this example, you should try to print of all 13 cards of a given suit and then all 52 cards, an entire deck.

Approach

Armed with the knowledge of graphic characters and their values, let's place this information into a table with variable names.

* see examples #2 and #3, which use `data` and `read` keywords

character table	variable	value
upper-left corner	ul	188
upper-right corner	ur	189
lower-left corner	ll	190
lower_right corner	lr	191
vertical bar	vb	130
horizontal bar	hb	150
card suit (heart)	cs	252

Jumping ahead, here is the code. On the next pages, we will discuss it, in depth:

```
10  ul=188:ur=189:ll=190:lr=191:cs=252
20  up=16:rt=6:dn=14:vb=130:hb=150
30  width=7:height=9:cls

40  cprint chr$(ul);:draw_chrs(width,hb)
50  cprint chr$(ur)
60  for y = 1 to height-2
70    cprint chr$(vb);
80    draw_chrs(width,32)
90    cprint chr$(vb)
100 next
110 cprint chr$(ll);
120 draw_chrs(width,hb):cprint chr$(lr)

130 for x = 1 to height-1
140   print chr$(up);
150 next
160 print chr$(rt);"A";:cprint chr$(cs);
170 print chr$(dn);chr$(dn);chr$(dn);
180 cprint chr$(cs);:print chr$(dn);chr$(dn);
190 print chr$(dn);:cprint chr$(cs);"A":end

500 proc draw_chrs (n, char)
510   for x = 1 to (n-2)
520     cprint chr$(char);
530   next
540 endproc
```

New keywords and techniques

Card uses `for`/`next` loops and `proc`/`endproc` to define procedures. If you've not worked with loops, they control program flow by iterating an index from a starting value to an ending value with *worker* keywords between. We used them in our character graphics work on the prior pages.

In this example (lines 60-100), we will use a loop to `cprint` a sequence of characters *y*-times. Elsewhere, we use loops to `print` a series of cursor up movements. Alternatively, we could have strung consecutive `chr$(char)` statements together, but looping is easier to read and understand.

This program also introduces `proc` and `endproc`. Used to define a procedure, this pair of keywords can be thought of as a subroutine that supports passed variables (in this case, the number of times to iterate and the character to `print`). We could have *hard coded* the data within, but by making it callable *with values*, we have the opportunity to re-use it for different purposes. One other great thing about a `proc` is the ability to `list` JUST the procedure, and do so by name, as in:

```
list draw_chrs()
```

empty parens are required

Card - Code discussion

The first section (lines 10-30) merely define constants and clears the screen. Short variable names are used here but you can choose longer and more meaningful names, if you desire.

```
10  ul=188:ur=189:ll=190:lr=191:cs=252
20  up=16:rt=6:dn=14:vb=130:hb=150
30  width=7:height=9:cls
```

The second section of code is responsible for drawing a rectangle on the screen using rounded corner characters and bar characters. It does so based on `width` and `height` variables which represent the dimensions of the card. If you look at lines 40-120 closely, you'll see 3 calls to `draw_chrs`.

On line 40 and 120, we draw a horizontal bar, and on line 80, we pass in ASCII value 32, otherwise know as whitespace `' '`.

```
40  cprint chr$(ul);:draw_chrs(width,hb)
50  cprint chr$(ur)
60  for y = 1 to height-2
70    cprint chr$(vb);
80    draw_chrs(width,32)
90    cprint chr$(vb)
100 next
110 cprint chr$(ll);
120 draw_chrs(width,hb):cprint chr$(lr)
```

variable name `ll`, as in "lower-left), not the #11

Line 40 begins by printing the upper-left corner with a semi-colon directive. It then calls the `draw_chrs` proc, passing in `width`, and the `hb` (horizontal bar) character. You might notice that when `draw_chrs` prints (see pg. 21), it also uses a semi-colon. This nuance is important because it needs to be flexible to support the next char in the loop or in cases where we want to print a 'closing' char, outside the proc, such as on line 50.

The loop from 60-100 prints `height` minus 2 number of lines with the vertical bar (`vb`) on the left and right side. Lines 110 and 120 do the same thing as lines 40 and 50, except using the lower-left (`ll`) and the lower-right (`lr`) corner and `hb`.

There you have it, twelve lines of SuperBASIC code (not counting the `proc` on line 500) to draw a box of any width and height (defaulted to 7 and 9).

What else to try? The routine above depends on a clear screen and then assumes the card will be printed from column 1. This is fine for this simple example, but in real-life use, you would want the option to print the card at any location on the screen.

To do so, you might choose to seek to a starting screen position prior to line 40, and then prefix printing between lines 60 and 70 with *n* number of cursor right characters or use the new `at`* directive to the `print` keyword.

* `print at {y},{x}; {text}` is supported as of the 1.1 release

Don't be afraid to experiment. Sometimes, attempts to modify code goes sideways, and since there is no 'undo' in the screen editor, you will need to get into the habit of saving your work frequently. In time, working on these types of routines will be second nature.

Lines 130-190 are responsible for printing the card suit data, but again, since this is meant to be a simple example, you'll find that it really only works for a given scenario; one with a dimension of 7 x 9, else, suit and values will not be centered.

This is the least elegant portion of the program. It starts by printing (using the actual `print` keyword, not `cprint`) a series of <CURSOR UP> characters. But then it prints the card face data using a brute force (hard-coded) approach:

- step one character over the assumed left vertical bar
- print the letter "A", then the card suit (heart) character
- print 3 down cursor chars
- print another card suit heart (this one will be in the center of the card)
- print 3 more cursor down chars
- print the heart again, and finally, the letter "A"

The BASIC keyword `end` terminates the program, else it would try to execute the `proc` definition and cause an error.

```
130 for x=1 to height-1
140   print chr$(up);
150 next
160 print chr$(rt);"A";:cprint chr$(cs);
170 print chr$(dn);chr$(dn);chr$(dn);
180 cprint chr$(cs);:print chr$(dn);chr$(dn);
190 print chr$(dn);:cprint chr$(cs);"A":end
```

What else to try?

It would be useful to relocate this code into its own `proc` and to accept card values and print colored suits. As above, it would also be good if this code worked for a card printed anywhere on the screen and for any size card.

In the early days of home computing, card games were very popular. Commodore platforms in particular (beginning with the PET), excelled in character-graphic heavy games.

Given a little work, it is feasible to port one of the early games to the Wildbits/K2 platform. But you are likely to find that early programmers did not have the discipline or the tools available today, and as a result, code is sometimes difficult to follow.

It's fun to take an existing program and work hard to port it to a different platform, but your time and effort may be better spent challenging yourself to write a new program from scratch.

Ok, now let's finish this up by discussing the `draw_chrs` proc:

```
500 proc draw_chrs (n, char)
510   for x = 1 to (n-2)
520     cprint chr$(char);
530   next
540 endproc
```

It may be obvious since we discussed its use above, but line 500 defines the procedure and identifies the variables that will be passed in. Notice that `n` and `char` are used within the body of this procedure and not in the main program. It is a common practice to refer to variables within procedures generically; that is what is being done here.

The loop on line 510 terminates at `n` minus 2. This is to accommodate the left and right side of the card or the corner characters (which are printed outside of the `proc`).

A word on SuperBASIC formatting

You may have noticed that some SuperBASIC keywords (when listed) are auto-indented. `proc/endproc` and `for/next` are two such sets of keywords (there are others).

Auto-indenting greatly enhances readability and you don't have to do anything to make this happen; simply type your line number and keywords with a single whitespace character between and SuperBASIC will take care of the rest!

Example #2 - Invaded (Redefining characters)

Everybody has heard of Space Invaders by Taito. It is possibly the second most recognized vintage video game, next to Pacman, by Namco.

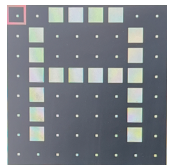
In this example, we will use redefined characters to transform a small set of unused characters into six segments of a familiar Space Invaders foe, and then put it into motion using a series of `peek`/`poke` keywords to manipulate *screen memory* directly.

We will also use the SuperBASIC `sound` keyword to coheres one of the K2's sound chips into making some low frequency sounds that mimics the alien action from the original game.

Graphic data consumes a lot of memory, especially in the context of sprites, tiles, and bitmaps. But since character graphics are two-colors (foreground and background), we will only need 48 bytes.

This equates to 6 characters; 3 on the first screen row and 3 on second, with each character requiring 8 bytes of data; that brings the total to 48 bytes (not too much typing!). We will use `data` and `read` keywords along with `poke` to define and overwrite a few built-in characters with custom graphics.

Let's study the attributes of a standard ASCII character (the letter 'A'):

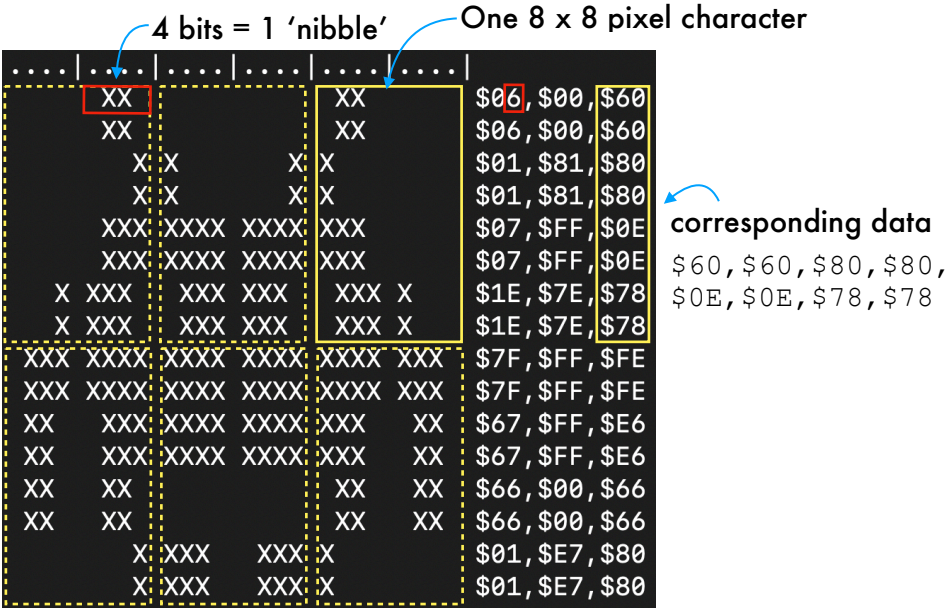


The gap on the left, right, and lower edge of this 8 x 8 grid maintain separation between characters when printed. Square boxes represent 'on' bits and small dots represent pixels that are transparent. (this image was

captured from **Font Editor***, by Ernesto Contreras)

Of course, we are creating a large, multi-character *shape*, so the gap between the characters is not desired; (we desire the opposite effect).

Translating an image from graph paper to data is an advanced topic and will not be covered in this guide, but here is the map of how the desired shape maps to bits which arrive at the values that we will be using:



What about Color

This program uses color simply. We will use a green on black motif, just like the original game.

Changing character color attributes is easy. You simply `print` the `chr$(x)` of a color value.

* **Font Editor** is part of the Wildbits Graphic Toolkit and is a character set editor written in SuperBASIC for your machine.

Selecting a color

Above, we mentioned there are `chr$` codes to control cursor movement, clear screen, and perform other functions, such as color. Each character may have one foreground color and one background color and there are 16 to choose from occupying `chr$` codes 128 to 160. Here are the highlights:

- Foreground color codes begin at a value of 128; there are 16 foreground colors to choose from (see chart below). You may choose to redefine your own color palette; check the reference guide for additional detail.
- Background color codes begin with a `chr$` value of 144; there are 16 dedicated background colors, as well. While vintage computers had a single background color for the entire screen (sometimes exercising trickery for two), the K2 can be thought of as having a choice of color for each of the 4,800 character positions on the screen (80 x 60).
- To select a green foreground color, we add a value of 3 to 128 and `print chr$(131)`.
- In this example (and in our program), everything printed after the keyword will be green until we change it again.
- To make the background black, we add a value of 0 to the base of 144, so: `print chr$(144)`.

Here is the full default color chart. By default, the background and foreground colors are identical, but you can modify this yourself by poking values into the proper memory map location:

0	black	8	dark gray
1	medium gray	9	light gray
2	blue	10	purple
3	green	11	light green
4	magenta	12	pink
5	brown	13	red
6	orange	14	yellow
7	light blue	15	white

Peeking and Poking memory

Your K2 computer has memory mapped input/output, which is another way of saying that devices 'live' at specific addresses.

It is desirable (and often necessary) to be able to *get* or look at a value from some memory address, or to *set* or change data at a specific address.

We use the SuperBASIC `peek` keyword to *look at,* or retrieve data and the `poke` keyword to modify data.

Everything in your computer plugs into memory one way or another, and this includes the screen, the keyboard, the LEDs that indicate CAPS LOCK, audio devices, everything.

In this program, we will be directly modifying screen memory to *move* the alien at the top of the screen from left to right.

To move (redraw) our alien, we need to know which character (or which part of the alien) is present at a given location, so we can move it to the next position (to the right). We also need to clear the space the alien last occupied by writing a `chr$(32)`.

All of this is to say that the core routine of this program is a bunch of `peek` and `poke` keywords that directly alter the first line of screen memory (address `$c000` - `$c04f`) and the second line of screen memory (address `$c050` - `c0af)`. It is not important that you understand this in order to run this program or to modify it, but at some point, you'll benefit from having this knowledge as a tool in your toolbox to help control other aspects of your K2 computer.

Using Sound

SuperBASIC has a handful of dedicated sound keywords such as `zap`, `shoot`, `ping`, `explode`, try them! There is also a keywords to play frequencies for a length of time on specific PSG channels; this programming example introduces `sound`.

The usage is straightforward, however, there is a bit of math required in order to map frequencies to western scale tones. See the SuperBASIC manual for more on this topic.

We will be choosing *atonal* (not a known musical scale) beeps that sound something like the famous Space Invaders 'marching' theme. You will know it when you hear it.

Now, let's move on to the code!

Approach

This program has 5 sections:

- initialization and a call to data load
- `poke` keywords to place the alien on the screen
- main loop (for animation)
- a `proc` for data load and another for alien stepping; the stepping function also manages sound and the time delay between steps
- six lines of data with 8 bytes per, which defines the character graphics bitmap

Here is the full code for "Invaded":

```
10  print chr$(131);chr$(144):cls:freq=800
20  load_chardata()

30  poke 1,2
40  poke $c000,1:poke $c001,2:poke $c002,3
50  poke $c050,4:poke $c051,5:poke $c052,6
60  poke 1,0

70  for x = 0 to 71
80    step_alien()
90  next: explode:explode:explode:end

500 proc load_chardata()
510   poke 1,1
520   for x= 0 to 47
530     read byte
540     poke $c008+x, byte
550   next
560   poke 1,0
570 endproc
```

```
600   proc step_alien()
610     poke $c000+3+x, peek($c000+2+x)
620     poke $c050+3+x, peek($c050+2+x)
630     poke $c000+2+x, peek($c000+1+x)
640     poke $c050+2+x, peek($c050+1+x)
650     poke $c000+1+x, peek($c000+x)
660     poke $c050+1+x, peek($c050+x)
670     poke $c000+x,32
680     poke $c050+x,32
690     for y=1 to (2500-(x*30)):next
700     sound 1, freq,4
710     freq = freq + 60
720     if freq = 1040
730       freq = 800
740     endif
750   endproc
800 data $06,$06,$01,$01,$07,$07,$1e,$1e
810 data $00,$00,$81,$81,$ff,$ff,$7e,$7e
820 data $60,$60,$80,$80,$e0,$e0,$78,$78
830 data $7f,$7f,$67,$67,$66,$66,$01,$01
840 data $ff,$ff,$ff,$ff,$00,$00,$e7,$e7
850 data $fe,$fe,$e6,$e6,$66,$66,$80,$80
```

New keywords and techniques

We covered most of the intricacies in the prior sections, but from a style perspective, there are a few things to talk about.

As programs get longer (or as you modify examples), you will inevitably encounter bugs and code that may not 'throw' an error, but also may not do exactly what you want.

Keeping your code *clean* will go a long way towards helping you resolve issues sooner, rather than later.

Liberal use of `proc`/`endproc` and a 'structured' manner of coding is best. Structured programming compartmentalizes code, ensuring that once tested, details are no longer important. You'll always know, for example, that calling `draw_chrs` will print `n` consecutive `char`.

The approach to be avoided mirrors 1970s practices that by necessity, let `goto` and `gosub`/`return` rule the day. Despite good intentions, these keywords require hard coded line numbers to direct program flow and this creates scenarios where renumbering becomes common and flow control becomes messy (creating what is sometimes referred to as spaghetti code).

Structured programming avoids the use of `goto`. Line numbers still exist, but only to maintain order and provide a means of reference. This makes renumbering much easier.

Another technique that is encouraged calls for a limited number of keywords per line. Of course, this contradicts the desire to keep your program limited in total length as we are now touting; ultimately, you will find that maintaining something such as:

```
510   poke 1,1
520   for x = 0 to 47
530     read byte
540     poke $c000+x, byte
550   next
560   poke 1,0
```

… is far easier on the eyes than this:

```
510   poke 1,1:for x = 0 to 47:read byte:poke
$c000+x,byte:next:poke 1,0
```

This style of coding (aided by auto-indent) is easier to maintain should you need to add functionality. In the structured example, you can see and consider each keyword in relation to other code, especially useful when evaluating nested conditionals. In the long, single-line example, it can be challenging to understand what the core of the routine is doing, and as a practical matter, you will find yourself encroaching on the 80 character line limit. (line 510 is 67 characters long!)

Something else to suggest is to be wary of the power of `poke` and to wield its power carefully. Since it directly alters

memory, there is no opportunity for an 'undo/redo'. A mistyped poke can freeze the machine or if you are lucky, will render the display difficult to navigate. The message here is to a) save your work frequently and b) take care in typing and understanding what you are doing (double check addresses and values).

You won't harm your machine, but you'll lose time having to re-type code or recreate hours of progress.

Invaded - Code Discussion

On with the code… the first section (lines 10-20) set the foreground and background color, clears the screen, and defines the base frequency for the `sound` keyword. As we will see, `freq` cycles between 4 different frequencies.

Line 20 calls a character bitmap load routine, discussed below.

```
10  print chr$(131);chr$(144):cls:freq=800
20  load_chardata()
```

Lines 30-60 place the alien on screen at location (0,0), but first, it pokes a value of 2 into the MMU I/O control register. This is an advanced topic* but the short explanation is, the lower 2 bits of this register control which of the four banks of I/O will sit in the `$c000-$dfff` memory range. A value of 0 is the default, a value of 1 is used in `load_chardata()` to swap in the font memory; and a value of 2, swaps in screen memory.

Location `$c000` represents the upper left corner of the screen (0,0). With 80 characters per line, the beginning of the 2nd line, in memory is `$c050`. Line 40 places the upper half of the alien on the screen and line 50 renders the rest. Line 60, returns the MMU I/O control register to 0, the desired default.

```
30  poke 1,2
40  poke $c000,1:poke $c001,2:poke $c002,3
50  poke $c050,4:poke $c051,5:poke $c052,6
60  poke 1,0
```

* see pg. 78 for more on memory management and the MMU I/O control register, specifically

Lines 70-90 contain the main loop which runs from 0 to 71 (steps across the screen), calling `step_alien()` each iteration. When the `for`/`next` completes, the program ends.

```
70  for x = 0 to 71
80    step_alien()
90  next: explode:explode:explode: end
```

The load routine is called once, but it made sense to move it to the bottom of the program and wrap it in a `proc`, for clarity.

After selecting the font I/O bank, a 48 step `for`/`next` loop reads data from lines 800-850 and stores it into font memory, beginning at `$c008`, indexed by *x*.

It can be tedious and challenging to enter large amounts of data without errors; sometimes, it is unavoidable. If your invader looks peculiar, double-check the values. An alternative is to use binary data from a file and then `bload` it into memory. This approach has challenges as well.

```
500 proc load_chardata()
510   poke 1,1
520   for x= 0 to 47
530     read byte
540     poke $c008+x, byte
550   next
560   poke 1,0
570 endproc
```

Lines 600-750 do all of the real work and is responsible for moving the alien (lines 610-660), cleaning up the 'tail' between 670-680, and managing timing, tone play, and frequency value shift. Now let's talk about the time calculation on 690.

With the 'outer' loop (lines 70-90) running from 0 to 71, each step of *x* calls `step_alien()` to move one horizontal position (a full character space) and invoke a delay of 2,500 minus the product of *x* and 30. As always, there are a number of algorithms that can be used; this one just seemed simple enough to implement and achieved the desired effect.

Lines 700 triggers a sound and 710-740 cycles the frequency value (causing lower tones) increased by 60, each step. On line 720, an out of range value of 1040 is checked for, and if found to match, `freq` is reset to 800, the starting pitch.

```
600 proc step_alien()
610   poke $c000+3, x, peek($c000+2+x)
620   poke $c050+3, x, peek($c050+2+x)
630   poke $c000+2, x, peek($c000+1+x)
640   poke $c050+2, x, peek($c050+1+x)
650   poke $c000+1, x, peek($c000+x)
660   poke $c050+1, x, peek($c050+x)
670   poke $c000+x, 32
680   poke $c050+x, 32
690   for y = 1 to (2500 - (x*30)): next
700   sound 0, freq, 4
710   freq = freq + 60
720   if freq = 1040
730     freq = 800
740   endif
750 endproc
```

voice 0, 1, and 2 generate pulse wave tones and voice 3 is noise

the `sound` keyword supports a 2nd form which introduces modulation*

One more thing about `sound`

The first parameter to the `sound` keyword (value 0) selects the voice type; in this case, a pulse wave. The 3rd parameter (value of 4) is duration and prescribes the # of ticks (at 60 or 70 Hz).

Referring to the SuperBASIC manual, you'll see that `sound` is asynchronous; audio is played at 111,563/*pitch_val* and queued (control returns to the program). The queue of tones will continue to play until complete, even if the program has ended.

This asynchronous behavior is handy for music applications or situations where you desire ambient background music, but it can be challenging to coordinate sound with visual cues of a game or a demo. Experiment with the `playing(0)` keyword; it returns the status of a voice (*true*, if playing / *false* if not).

* **Getting Started Volume II** covers K2 audio capabilities in depth

Example #3 - Balloon gone WILD (Sprite graphics)

In 1982, Commodore released the most successful home computer in history, the Commodore 64.

Its success can be thanked, at least in part, to a proprietary chip called the VIC-II 6567 which allowed the programmer to define and manipulate up to 8 color movable objects called “sprites”, with ease.

The “User’s Guide” that accompanied the computer included a 16 line demonstration program (see page 71 of the guide here) that changed the world.

Titled “UP, UP, AND AWAY!”, it defined a single-color sprite in the shape of a hot air balloon, set values to instantiate it on the screen, and then moved it diagonally across the screen in an endless loop.

Millions of children and adults worldwide, typed in the 16 lines of BASIC code and watched in wonder, then thought to themselves, “I did that!”.

This feat, to create a [then] high-resolution graphic object on the screen and smoothly move it at a relatively rapid rate on a 1 MHz computer was not possible prior, at least not with such ease.

Our example #3, “Balloon gone WILD”, is both based on “UP, UP, AND AWAY!”, and dedicated to its impact on a long list of Gen-X kids and their families and friends that were exposed to it.

It’s hard to imagine at this point, but in the late ‘70s and early ‘80s, home video games were no more complex than a player controlled object, a missile object, and one or more adversaries that in the best of cases, had some still graphics and rudimentary

sound effects. The advent of the sprite and a modest amount of code was enough to put the power of *arcade game play* into the hands of mortal humans, (young girls and boys) born at the right time and place to experience the dawn of technology that later generations would take for granted!

This power is now in your hands

Did you know that your K2 computer supports, not just 8 sprites, but 64 (and soon, 128). Each may be full color and up to 1,024 pixels (32 x 32) in size. Colors are defined with 24-bit RGB depth and plugged into one of many custom color palettes.

Our third example is only possible because of the benefit of time (40+ years) and the dedicated and talented individuals that worked hard to pack an outrageous amount of technology into increasingly smaller and more affordable components that have come together in your computer, your K2.

In this example, we will define a multi-color balloon and add a dose of randomness to control its ascent and landing. We will also leverage rudimentary color palette rotation in order to enhance the animation.

Before we get there, we have a lot of ground to cover, and we'll approach it in stepping stone fashion with a handful of demo programs and a study of data. By the time we get to the main program, you'll be ready to create something of your own design.

New keywords and techniques

We will also talk some more about *clean code* and look at examples that demonstrate what to do and

what to try and avoid doing. (Disclaimer: we break a few rules for the sake of performance and brevity here and there)

“Balloon gone WILD” is broken into two parts; one that sets up the sprite image using a loader with 1,024 byte values of shape data (in `data` statements), and another that runs the actual demo in an endless loop.

The K2 supports four sprite sizes: 32 x 32, 24 x 24, 16 x 16, and 8 x 8. Now is a good time to mention that the 8 x 8 sprite is not the same size (on the screen) as the 8 x 8 pixel text characters we discussed above; they are 4x as large by area.

This is due to the 320 x 240 graphics resolution of your K2 system (for bitmaps, tiles, and sprites), and the 640 x 480 resolution for text and graphic characters. If you commingle, you’ll need to keep this translation in mind. As you learn more about the system, you will most likely be writing code that benefits from this difference.

What makes a sprite, where to start?

Sprite definitions require 4 attributes and there are keywords and techniques to assist. They describe:

A. which **image** or shape (by number) should be used
B. where on the screen (*x*, *y* **location**) to position it
C. what **color** palette *definition* to use
D. what sprite **size** matches the data

But first, we need to enable (turn on) sprite graphics mode:

```
sprites [at {addr}] on
```

This keyword enables graphics mode and turns the sprite engine on; it also turns the screen black. We only need to do this once; if you have the need, you can turn `sprites off`. The optional `'at'` parameter will alter the base memory location, but we will use the default: $03:0000 aka $30000.

The only other keyword of consequence is `sprite`, and it will let you configure attributes A (*i*) and B (*x*, *y*) as follows:

```
sprite {n} image {i} to {x}, {y}
```

n = the **sprite number** and can range from 0..63. SuperBASIC needs to know which sprite we are manipulating.

i = the ***image* number**[A] should not confused with the sprite number; it is an index into image/shape memory by count.

Multiple sprites can be associated with the same image data. A good example of this is the famous Space Invaders arcade game and specifically, the aliens. There are only 3 bad guys (not including the UFO). So you need only to define 3 shapes; data is reused across rows and columns for the 55 aliens that we are accustomed to seeing at the start of a level.

Most of the data and attributes associated with sprites are stored in dedicated memory locations called 'registers' which are tables tied to hardware, but you do not need to be concerned about this. SuperBASIC manages most of it for you.

The **sprite location**[B] as ***x* and *y*** is expressed as coordinates (x ranges from 0..319 and y ranges from 0..239). Placement of a sprite within SuperBASIC considers *the center of the sprite, its anchor point*.

If you've worked with K2 sprites in assembly language, C, or with a direct `poke` in SuperBASIC, this will be different from what you are accustomed to (as far as the machine hardware is concerned, a sprite is anchored by it's upper left corner).

SuperBASIC considers the upper left of the screen as 0, 0 and the lower right as 319, 239.

Two attributes we have not yet discussed

The K2 arranges 255 colors into a list called a palette (akin to a painters palette). Each color definition is composed of red, green, and blue components and each can range in value from 0-255. This is what is known as 24-bit color (and it maps well to Pantone colors). **Sprite color**[C] is determined on a per-pixel basis, meaning, each and every pixel can be any of 255 colors, or be transparent (color 0).

SuperBASIC comes pre-loaded with a full palette of colors which are free for us to use without any wrangling; sometimes (as we will explain below), it is advantageous to set and manipulate the color palette. And despite machine support for multiple palettes (each with 255 colors), SuperBASIC focuses solely on palette #0, so keep this in mind.

The last thing to discuss is **sprite size(D)**, and again, we will need to tell the sprite engine which size we want a each sprite to be. The four sizes are numbered from 0 (smallest, the 8 x 8) to 3 (largest, the 32 x 32). Our sprite uses type 3.

In addition to the obvious effect of visually sizing the graphics object, you need to be aware that an incorrectly set shape value can really mess up the way your sprite looks AND how other sprites in your 'sprite set' are interpreted.

This too is an advanced topic and will be discussed as part of "things that can go wrong", but in short, the size of a sprite affects how memory will be interpreted in terms of the "stride" or width of the graphic object.

There is one other feature worth mentioning, it's the `hit()` keyword. It tests for collisions between sprite objects using an algorithm called the "box test". We will not be using it in any of our examples, but have a look at the SuperBASIC manual and experiment, to learn more.

Next we discuss *creation and management*, otherwise known as "how do we design a sprite, what format does it need to be, and how do we get it into memory and moving".

The next part is challenging, but also fun. So grab a snack and a beverage and let's go!

Platform tools, and the absence of them

The de facto sprite manipulation tool is the Wildbits Sprite Editor ('spredit.bas') written by Ernesto Contreras. Ernesto wrote an entire suite of graphic manipulation tools and you can find them on the SD card that came with your system.

They include a character/font editor ("font.bas"), and tile editor ("mapedit.bas") in addition to a pair of transmute tools.

Spredit (the preferred, and *first* method to create a sprite) is joystick (or mouse) driven and supports sprite features in addition to workflow tools for cloning (for subsequent editing), useful when creating multi-frame sprites (for animation). It also has tools for palette manipulation save and load, the ability to mirror *x* or *y*, and more.

Step 1 in getting a sprite up and running on the K2 is to create a sprite in the first place. In most cases, you'll start with a blank slate and an idea.

A *second* method (no tools required) is to convert from an encoded string format (we offer samples below) that supports in-program editing and uses `data` statement and a load loop.

Still, there is a *third* method, which is the old fashioned way; graph paper and calculated values punched in to data statements as base-10 or base-16 (hexadecimal) numbers. The Commodore User's guide discusses this method, applicable to the C64. Of course the K2 is a full color machine, so we need to manage bytes of color, not bits of pixels, but conceptually, it's a similar exercise. Let's begin with the second method.

2nd method: Encoded string data

The program on page 42 does everything discussed thus far and can be thought of as an MVP, otherwise known as a "minimum viable product" or "program". We will refer to this as "`one invader.bas`". If you have a eye for vintage arcade, you might recognize the encoded graphic (see lines 2000-2007).

This program also uses the `memcopy` keyword in conjunction with memory at $7800 to transfer from an area SuperBASIC can `poke` and `peek` to, into upper memory. (lines 60, 90, 110, and 150-180 take care of the calculation and memory manipulation). We will discuss this and SuperBASIC's special "sprite data set" format in detail on pages 45-47.

```
10   sprites on:size=0:pal=0
20   xfer_addr=$7800: col=24
30   for y = 0 to 7
40     read line_data$
50     for x = 0 to 7
60       addr=xfer_addr+(y*8)+x+3
70       pixel$=mid$(line_data$, x+1, 1)
80       if pixel$="."
90         poke addr,0
100      else
110        poke addr,col
120      endif
130    next
140  next
150  poke xfer_addr,$11
160  poke xfer_addr+1,size
170  poke xfer_addr+2,pal
180  memcopy xfer_addr, 67 to $30000
190  sprite 0 image 0 to 100,100
1999 :
2000 data "...oo..."
2001 data "..oooo.."
2002 data ".oooooo."
2003 data "oo.oo.oo"
2004 data "oooooooo"
2005 data "..o..o.."
2006 data ".o.oo.o."
2007 data "o.o..o.o"
```

sprite color, which points to palette entry #24 (green)

if ".", a zero will be poked for an 'off' pixel (transparent)

otherwise, the single color defined by `col` (green) will be poked

make sure there are 8 rows

'stride' should be 8 also; make sure every line has 8 characters

If you typed this in, save it before you attempt to run it! It includes several `poke` statements and a `memcopy` so there is plenty that can go wrong.

If your typing skills were up to par, running the code will create your small green friend and place it at 100, 100. After all of this setup, line 190 makes it happen!

You might wonder, how the use of a sprite differs from the larger invader in the “Invaded” example on pg. 26. The code and methods are different, but how? Here is a quick look:

	“Invaded” char example	**“Last Invader” sprite**
Graphic feature	Redefined characters	Sprite graphics
Encoding	Binary (bit per pixel, which is monochrome)	Byte-wise (8-bit byte per pixel, indicating color number)
Color potential	Monochrome but may have background (aka ‘BG’) color	Choice of 255 (per pixel basis and allows for transparent BG)
Movement method	Poke 8 chars to screen memory (a 3 x 2 image)	`sprite` keyword or `poke` of registers
Movement granularity	Per char space (about 4 ‘fat’ pixels)	Per fat pixel (1/4 of a character space size)

While this is interesting, the table above does not truly capture the benefits, which are numerous: ease in managing color, orchestrating shape swapping, supporting cleaner and fewer lines of code, but most importantly, getting more work done in less time.

Let’s try two quick things before moving on. Assuming your sprite is still on the screen, type the following in immediate mode:

```
for x = 0 to 319:for y = 1 to 75:next:sprite 0
image 0 to x, 100:next <press RETURN>
```

single line

Compared to the “Invaded” code (lines 610-680 on pg. 35), there is very little required to move the object so the possibility of moving dozens of sprite objects around in the same amount of time is feasible. You can also have sprites cross over each other without worrying about the complexity of managing otherwise destructive writes to screen memory.

On the next pages, we'll examine how to do more with very little code.

Diving deeper… into color

Carefully modify the following lines and type `run` again.

```
110  poke addr, asc(pixel$)
2000 data "...--..."
2001 data "..bbbb.."
2002 data ".++++++."
2003 data ";;.;;.;;"
2004 data ";;;;;;;;"
2004 data "..+..+.."
2006 data ".g.66.g."
2007 data "<.1..1.>"
```

`asc()` converts a character to a byte value; we used the `chr$()`, the opposite, above

unrecognizable image?

If you typed the `data` statements correctly (or even if you did not), you should see the same small alien image as the green example above, but with a colorful striped skin.

Look closely at the individual pixels on the screen; in particular, examine the lower-left and lower-rightmost pixels of the alien and the corresponding data. Line 2007 may seem pleasing from a symmetry perspective, but these two characters '<' and '>' represent two adjacent entries in the color palette. By default, they are just a few degrees apart in color. Can you tell?

We could have redefined these, one as RED and one as WHITE, but these are the default colors; we'll dive into the intricacies of color shortly.

Encoding pixel/color bytes in this way has little to do with SuperBASIC. This was just a fabricated example with an unusual method for encoding color within data. It merely translates a pixel's desired color into a value from 1 to 255 and punches a byte into memory. Said another way, the ASCII value of each non-"." character identifies the color number of a pixel. Positions which are "." (as on pg. 42), point to the 0th index of the palette (transparent).

Now go ahead and re-run the spite move code on pg. 43. You'll notice that it runs exactly the same as when the sprite was solid green.

This would not be true of redefined characters. In example #2, we redefined characters and wrote direct screen pokes to move them but did not consider color (we changed the text background and character color once).

Dealing with character colors on an individual basis requires a separate bank of data that is identical in size to that of screen character memory (4,800 bytes with two nibbles per character). So you might envision the character move code being twice as long if it had to deal with anything but a single color graphic.

The changes you made on pg. 44 to the sprite color merely modified the sprite definition in memory; attributes such as the colors used, size of the sprite and other data are stored as part of the image. So moving it (or a much larger sprite) is easy. It works identically to the original with no performance penalty.

The `sprite` keyword still refers to this particular sprite (#0, in this case) which we tie to image #0.

Meta-data definitions and the "Sprite Data Set"

The heavy lifting of this programming example is carried out between lines 30-140 and it is a classic loop within a loop.

The outer loop on line 30 (iterated by *y*) deals with rows of data and executes a `read line_data$` on line 40 to get the next row. The inner loop (on line 50 iterated by *x*) calculates the address, extracts an individual character from a row of data, and places it into temporary (aka transfer) memory.

Placing data into memory, however, is only half of the job. We still need to tell SuperBASIC how to interpret the sprite shape data and that is where *meta-data* comes in.

Generically, the term "meta-data" is defined as *data-about-data*, and here, it is just that. It answers the question "how do we describe this sprite image" and note, we are talking about

the *image* here, not the 1 of 64 sprites that can exist on the screen. This is *just* the image, and you can think of it as a template. In case you have not yet realized, you can have hundreds or even thousands of sprite image definitions… as many as memory allows.

SuperBASIC needs to know two things about each image: how wide is the image in pixels/bytes (aka the *stride*) and secondly, which color palette will be used (always #0).

Lines 160-170 take care of this in our single sprite example:

```
160  poke xfer_addr+1,size
170  poke xfer_addr+2,pal
```

As we've been hinting, the format supports a 'set' of sprite images. But how is the set delineated and how can SuperBASIC discern one from another, if the sizes vary?

This excerpt from the SuperBASIC manual explains:

6.5 Data format

At present there is a very simple data format.

```
+00 is the format code ($11)
+01 is the sprite size  (0-3, representing 8,16,24 and 32 pixel size)
+02 the LUT to use (normally zero)
+03 the first byte of sprite data

The size, LUT, and data are then repeated for every sprite in
the sprite set.  The file should end with a sprite size of
$80 (128) to indicate the end of the set.
```

This brings us to the last thing that we need to resolve; it's right there at the "`+00`" marker, above: the entire block of sprite image definitions needs to begin with a special token, a single byte of hexadecimal, value $11 (17 in decimal). And it must end with an $80 or 128 decimal in the spot where the next size would otherwise exist.

The following data example depicts a sprite data set with 2 images defined. The first is our 'small' alien, the second represents the 'large' alien from "Invaded" on pgs. 26-35. We

will be using a *medium* (16 x 16) sprite for the blue alien. If you look closely, you'll notice that the image is only 8 bits high, so the lower half is transparent. (but there is a gotcha!)

```
$03:0000     $11     (token that identifies the START of the sprite data set)
$03:0001     $00     (shape #0's size = 8 x 8)
$03:0002     $00     (color palette number of 0)

$03:0003     $00,$00,$00,$18,$18,$00,$00,$00
$03:000b     $00,$00,$18,$18,$18,$18,$00,$00
$03:0013     $00,$18,$18,$18,$18,$18,$18,$00
$03:001b     $18,$18,$00,$18,$18,$00,$18,$18
$03:0023     $18,$18,$18,$18,$18,$18,$18,$18
$03:002b     $00,$00,$18,$00,$00,$18,$00,$00
$03:0033     $00,$18,$00,$18,$18,$00,$18,$00
$03:003b     $18,$00,$18,$00,$00,$18,$00,$18

$03:0043     $01     (shape #1's size = 16 x 16)
$03:0044     $00     (color palette number of 0)

$03:0045     [$00,$00,$3b,$00,$00,$00,$00,$00
$03:004d     $3b,$00,$00,$00,$00,$00,$00,$00]
$03:0055     $00,$00,$00,$3b,$00,$00,$00,$3b
$03:005d     $00,$00,$00,$00,$00,$00,$00,$00
$03:0065     $00,$00,$3b,$3b,$3b,$3b,$3b,$3b
$03:006d     $3b,$00,$00,$00,$00,$00,$00,$00
$03:0075     $00,$3b,$3b,$00,$3b,$3b,$3b,$00
$03:007d     $3b,$3b,$00,$00,$00,$00,$00,$00

$03:0085     $3b,$3b,$3b,$3b,$3b,$3b,$3b,$3b
$03:008d     $3b,$3b,$3b,$00,$00,$00,$00,$00
$03:0095     $3b,$00,$3b,$3b,$3b,$3b,$3b,$3b
$03:009d     $3b,$00,$3b,$00,$00,$00,$00,$00
$03:00a5     $3b,$00,$3b,$00,$00,$00,$00,$00
$03:00ad     $3b,$00,$3b,$00,$00,$00,$00,$00
$03:00b5     $00,$00,$00,$3b,$3b,$00,$3b,$3b
$03:00bd     $00,$00,$00,$00,$00,$00,$00,$00

$03:00c5     $00,$00,$00,$00,$00,$00,$00,$00
$03:00cd     $00,$00,$00,$00,$00,$00,$00,$00
$03:00d5     $00,$00,$00,$00,$00,$00,$00,$00
$03:00dd     $00,$00,$00,$00,$00,$00,$00,$00
$03:00e5     $00,$00,$00,$00,$00,$00,$00,$00
$03:00ed     $00,$00,$00,$00,$00,$00,$00,$00
$03:00f5     $00,$00,$00,$00,$00,$00,$00,$00
$03:00fd     $00,$00,$00,$00,$00,$00,$00,$00

$03:0105-$03:0144  (64 bytes, all $00; not shown)

$03:0145     $80     (token that marks the END of the sprite data set !!, phew…)
```

This is an 8x8 sprite; defined by 64 bytes, each representing one pixel

stride of 16 bytes between '[' ']'

This is a 16x16 sprite; defined by 256 bytes, each representing one pixel. The 3rd group of bytes ($03:00c5 .. $03:0104) contains all $00 (transparent), as does the 4th group of 64 bytes ($03:0105 .. $03:0144)

The ‘gotcha’ is the larger (blue) invader is also 8 bits tall but needs 11 bits to accommodate its width; thus, we chose 16 x 16 (medium size).

Since we started with an 8 x 8 invader, it seemed appropriate to have the larger image defined within the top 8 scan lines and have the lower 8 transparent (pg. 47); but this turned out to be problematic.

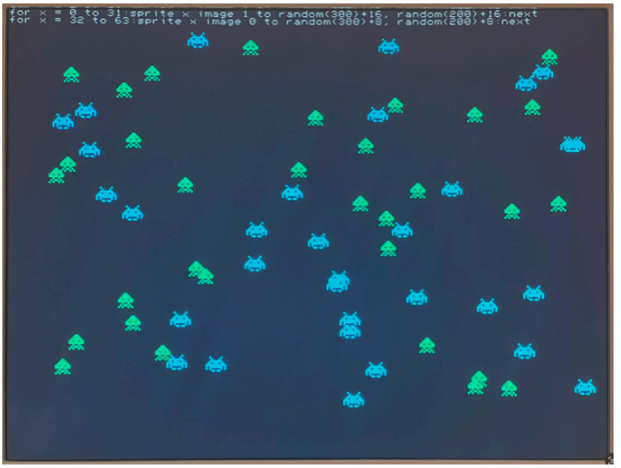
On pg. 39, we learned a sprite’s position is anchored at the ***center***. Defining it as we have would result in the two sprites appearing offset on the screen, despite an identical *y* coordinate. To fix this, we need to shift this sprite down within its field by 4 pixels. The sample program “`two invaders.bas`” does this; here is a look at the data.

```
2000  data "...oo..."
2001  data "..oooo.."
2002  data ".oooooo."
2003  data "oo.oo.oo"
2004  data "oooooooo"
2005  data "..o..o.."
2006  data ".o.oo.o."
2007  data "o.o..o.o"
2999  :
3000  data "................"
3001  data "................"
3002  data "................"
3003  data "................"
3004  data "..x.....x......."
3005  data "...x...x........"
3006  data "..xxxxxx........"
3007  data ".xx.xxx.xx......"
3008  data "xxxxxxxxxx......"
3009  data "x.xxxxxx.x......"
3010  data "x.x.....x.x....."
3011  data "...xx.xx........"
3012  data "................"
3013  data "................"
3014  data "................"
3015  data "................"
```

Added fun - bringing the *two invaders* example to life

Load and `run` the “`two invaders.bas`” program from the SD card distribution. It is a ‘setup’ program that instantiates sprites by sorting out memory and placing them on the screen. It’s similar to the prior “`one invader.bas`” example on pg. 42, except you’ll see that the load routine is adds the 16 x 16 sprite. The use of **`x`**’s is arbitrary; any character except a “**`.`**” will be poked into memory as color `col`.

If everything is correct, you should see two sprites on the screen, one from each image definition. To do something more fun with them, try the following:




```
for x = 0 to 63:sprite x image random(2) to
random(300)+8, random(200)+8:next
```

single line

You should see something similar to the following:
Now for added fun, press <CURSOR UP> then <RETURN>, repeatedly, as quickly as you can.

It's easy to take for granted that SuperBASIC, despite being an interpretive language, can redraw all 64 sprites, access two different shapes while generating 3 random numbers per iteration this quickly.

This is what sprites are all about. Data organized in shape tables, supported by specialized hardware to do work that would have required a significant amount of CPU cycles and memory access if done in software, alone.

Stride and strife - things that can go wrong

There are plenty of things that can cause graphics to go wrong and when it occur, it can be disorientating and disappointing. Here are two scenarios and examples of SuperBASIC errors that you might encounter. This applies to tiles, and to some extent, bitmaps, as well (see next page):

- If your graphic data does not have a consistent and/or correct 'stride', you could end up with a partially shifted image that does not reflect your intended design. Double check your work; make sure your formulas are sound. It's always a good idea to check for off-by-one and edge-of-bounds errors, as well.

  To see this in action, remove a character from a string on line 2004 above, then re-run the loader; fun, right?

- Landing data in the wrong part of memory or inadvertently stepping on, or misaligning meta-data associated with a sprite can also cause an image to be rendered improperly. `memcopy` is powerful, but since it overwrites target data without discretion, a miscalculation can cause havoc.

  If you want to have some fun, you can intentionally cause corruption with a poke into the size byte of the first sprite, or step on the start or end byte (less fun). Do this interactively, then rerun the program to fix it.

There are safety measures built into SuperBASIC including one to prevent use of a `sprite` keyword for image data that is not properly defined. When definitions within the sprite data set are questionable, SuperBASIC will present a generic `Out of Range` error message.

In this case, SuperBASIC is guessing that you that you are attempting to use an image definition that does not exist. You'll also get this error if the start of the data set ($03:0000 by default) does not contain the format code of $11 (17).

Another common error will result if an *x* or *y* value goes negative; picture a joystick driven player object with which your code subtracts values from a variable when moved upward or to the left. If a value moves from 0 to -1 the `sprite` keyword will return an: `Illegal argument` error.

This does not occur on overflow, at least not at the edge of the screen, as you might expect. Internally, the *x* register tests for a 10-bit value so you can punch in a value up to

1,023. Beyond this, you'll error out with a "`Syntax Error`". The *y* register has no such bounds checking but you'll get a general "`Illegal argument`" error by using a value larger than 65,535 (the maximum value for a 16 bit number).

As you become more familiar with SuperBASIC, you will see this error message elsewhere, as well; such as trying to:

`cprint chr$(256)` (max. value of an 8-bit number is 255)

Moving on: How to see all of the graphics palette colors

On page 52-53, we will use 8 extra-large sprites to paint the screen from left-to-right with the entire graphic palette. All 255 values cannot be squeezed onto the vertical axis (which has 240 scan-lines, so we need to do some additional math to encode values so we can lay them out horizontally.

Graphic characters are used to mark breakpoints every 8 colors, from 0-255 (remember, color 0 is always transparent).

This program is good to keep around; it can help you visualize your own palette definitions. Absent a yet-to-be-developed graphics tool, you will be challenged to see the entire palette.

The Wildbits Sprite Editor packs a lot of power into one application and has good palette editing capabilities (including a mouse controlled GUI slider per R, G, or B, to choose individual values, load and save, etc.) but it is destructive since it overwrites the current palette upon startup.

If you've been following along, you probably recognize some of the variables and techniques used prior. However, this program uses a top-down structure with no `proc`/`endproc`, `goto`, or `gosub`/`return` statements.

All of the work is performed within a 3-deep loop which runs iterator *n* across 8 sprites; for each sprite, *y* indexes horizontal scan lines, and *x*, populates values on the line, repeating the pattern for all 32 lines of each sprite. Line 100 ultimately calculates the address from current iterator values and 110 pokes the color data into memory.

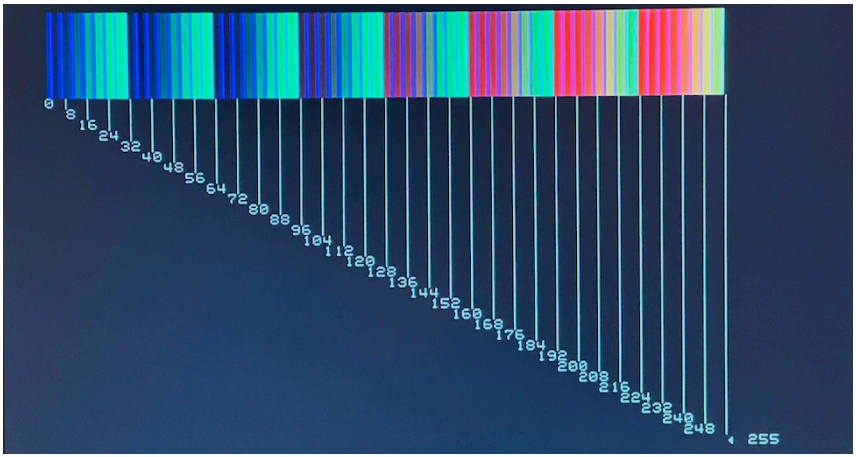
It would be easier to do this with bitmap graphics (which share the same palette), but we are discussing sprites so may as well invest some effort and do something challenging.

The first part of the program (lines 10-220) defines variables and then builds a sprite data set with 8 extra-large sprites, populating meta-data as we have done prior; it runs rather slowly due to thousands of calculations. The printing (screen scrolling) do not help performance, but the visual indicator is helpful to gauge progress.

In total, 8K (about 1/4 of SuperBASICs memory footprint) is moved up to the 03:0000 RAM bank, one sprite at a time.

After prompting the user to press <RETURN> with `input`, the sprites are displayed and a few more loops are employed to print a graduated index of decimal numbers; they mark “by-eight” breakpoints in the palette.

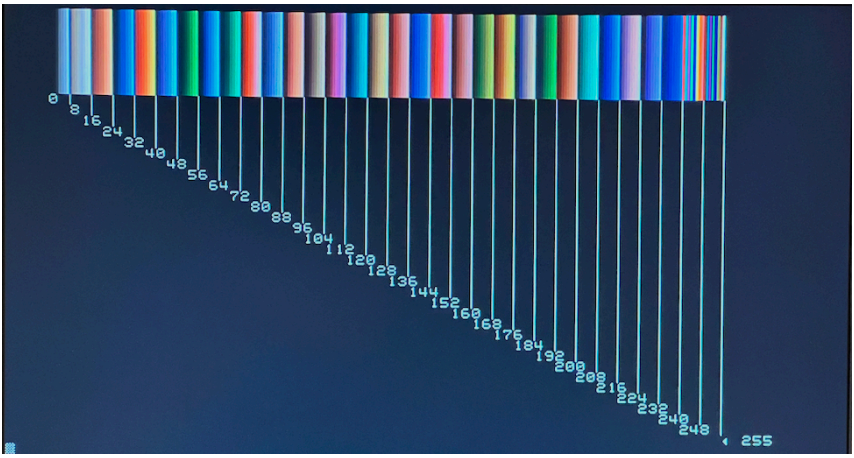
If you run with the default palette directly after a cold start you are likely to see the following:



There is some subtlety between colors but you’ll likely agree that this palette is missing plenty and not well organized.

As we will see in the next section, the Wildbits Sprite Editor is loaded with a custom palette which includes a well organized set of gradients, gray scales, and a set of simple colors in slots 239-255.

Here is a screenshot of the Sprite Editor palette:




```
10    col_base=0:xfer_addr=$7800:dest_addr=$30000:meta=2
20    sprites off :poke xfer_addr,$11
30    memcopy xfer_addr,1 to dest_addr:dest_addr=dest_addr+1
40    for n=0 to 7
50      print "sprite image #: ";n
60      print "color base    : ";col_base
70      for y=0 to 31
80        print "[";
90        for x=0 to 31
100         addr=xfer_addr+(y*32)+x+meta
110         poke addr,col_base+x
120         print ".";
130       next
140       print "]"
150     next
160     poke xfer_addr,3
170     poke xfer_addr+1,0
180     memcopy xfer_addr,1026 to dest_addr
190     dest_addr=dest_addr+1026
200     col_base=col_base+32
210   next
220   poke xfer_addr,$80:memcopy xfer_addr,1 to dest_addr
230   print :print :input "Press <return> to display color
palette";a$
240   cls :for x=1 to 9:print :next
250   sprites on
260   for n=0 to 7
270     sprite n image n to n*32+32,20
280   next
290   for y=0 to 31
300     for r=1 to 5-len(str$(y*8)):print " ";:next :print y*8;
310     for x=0 to 31-y
320       cprint " ";chr$(134);
330     next :print
340     for n=0 to y:cprint "  ";:next
350   next
360   cprint "    ";chr$(249);" 255"
```

Program structure, two parts

From a cold start (machine off), execute the following:

```
load "balloon gone wild setup.bas"
```

Running it loads the sprite data with a `data`/`read` loop and places it at 0, 0 (top left). This sprite is large, in fact, it's the extra-large variety (32 x 32 pixels or 1,024 bytes of data).

As a first step, let's examine the code that printed the text on the screen, leaving the blinking cursor at the end of the line. Whatever you do, do NOT press RETURN. Instead, do the following:

- move the cursor down several lines
- type `list 40, 70`

Can you figure out what it is doing?

```
40  cls
50  print "press RETURN and type 'Y' if asked"
60  print "load ";chr$(34);"balloon gone wild main.bas";
70  print chr$(34);
```

You may know that printing an ASCII 34 will display a quotation mark ("). It is not possible to print a double quote with the `print` keyword alone. We can print the `chr$` of the value of a quote, or we can `poke` screen memory directly (this is tedious); but anytime we use a quote in BASIC code, the interpreter treats the text that follows as a string literal.

This code places a complete and properly formatted `load` statement on the screen and leaves the cursor next to the closing quote. This puts the user in position to press <RETURN> to load the main program. We could have let the setup program hand off (load and run) the main, but wanted to demonstrate another use of <cursor up> and secondarily, pause the process of loading for a few exercises.

The sprite should be positioned partially off-screen, so let's fix this by moving it and then examine a new keyword to improve some of the colors you see. Now type:

```
sprite 0 image 0 to 100, 100 <press RETURN>
```

The balloon should have moved to mid-screen:

Now flip back to pg. 52 and study the **left** side (low numbered colors 1 .. 9) on the color spectrum. You most likely see shades of grays and blues. While not very attractive, consecutive palette slots were chosen for a reason, as we will discuss.

Since color data is not stored with the sprite image, care must be taken to either load system-ready `.pal` data into memory, or, to establish the referenced portion of the color palette in code. We chose the latter. But before we do, let's have some fun.

Our sprite balloon is built from colors #1 through #9 (plus #0, which is transparent); let's conduct a brief experiment.

Palette color #	RED	GREEN	BLUE
0 (transparent)	n/a	n/a	n/a
1	255	0	0
7	96	96	96
9	0	255	0

With some sample color data in the table to the left, we'll use some code to setup the palette.

To do this, we will use the `palette` keyword, which accepts four parameters; the **color number**, and then the **R**, **G**, **B** values, which are 8-bit numbers. They range from 0 (off/dark) to 255 (full on).

Step 1: zero out the palette with a `for`/`next` loop:

```
for x = 0 to 9: palette x, 0, 0, 0: next
```

The balloon will disappear, but don't panic; it's still there but will be BLACK (not transparent). Any time your object matches the screen color, it looks transparent, but if against a field of other graphic objects, you are likely to see parts of it.

Step 2: type the following, we are about to shine some light on this. Can you guess what color it is going to be?

```
palette 9, 0, 255, 0
```

Step 3: define part of the basket with the following:

```
palette 1, 255, 0, 0
```

Step 4: define color 7 for the rest of the basket and balloon:

```
palette 7, 96, 96, 96
```

color # **red** **green** **blue**

Fun RGB color facts:

- `0` in any color position turns that color component completely 'off', so: `0`, `0`, `0` = black. (which is not the same as it being transparent).
- a value of `255` turns that color fully 'on'; therefore, values: `255`, `255`, `255` equates to white. Think of turning the contrast and brightness of a CRT monitor all the way up.
- finally, an entry with the same color values across all three will be some level of gray, as in color #7 (`96`, `96`, `96`) above.
- ALL other possible colors are some combination of these 3 bytes. This is 24-bit color, otherwise known as 2^{24} or 16.7 million colors (you'd be challenged to discern even 16 bits or 65,536 colors from each other).

Finally try googling: `donkey kong blue rgb color code`

Depending on your search engine, you might get a few opinions, including `50`, `173`, `169`. So try it with:

```
palette 7, 50, 173, 169
```

It might not look great against the green '**W!**', but I think you will agree that the color handling on your K2 is amazing. And if commercial arcade is your thing, the internet is full of color data to help your projects look authentic.

Color has come a long way in 40 years. In 1986, the Amiga Home Computer was first to break the 256 color barrier, offering 4,096 hues. And when the famous bouncing ball was unleashed, the general public nearly lost their mind.

The next section discusses manipulation of the color palette as a form of animation. Our Balloon gone WILD example makes great use of this technique.

New technique: Palette cycling

It is common practice to 'flip' sprite images for animation purposes. Picture a hot air balloon popping and falling uncontrollably to the bottom of the screen, changing its shape a few times per second as it descends.

To accomplish this, you might leverage *spredit* to clone the original balloon, edit a copy, then clone the 2nd image and edit again. You might repeat this process several times.

After saving, you can use the *transmute* tool to convert the set to a load-ready file. Then (at the start of your program):

`bload "sprite set", $30000` and use `sprite` keywords with a static sprite number but different images as the balloon falls, animating it every 10 or 20 pixel change in *y*.

Examining a different use case, assume you wanted to simulate an animated flame from the bottom of the balloon. Similar to above, you could use several sprites, each with a different color scheme. But this would consume a lot of memory, especially at 1K per image (the size of a 32 x 32 sprite).

There is a more efficient way to manage this, especially when dealing with large bitmap landscapes, and that's where palette cycling comes in.

As we established, the `palette` keyword allows you to dynamically alter the RGB settings of a color slot. Any pixel encoded with that color number will change across the entire sprite. In the Balloon gone WILD example below, we animate the balloon's flame, cycle a green beacon at the top of the balloon, and randomly change the shade of the "W!"; and this is all accomplished with very little code.

If you'd like to experience palette cycling at a grand scale, have a look for the "Living Worlds" demo on the SD card that came with your K2 system; it uses palette cycling on a large bitmap.

Living Worlds was ported from Mark Ferrari's famous example into 6502 assembly language by Discord member Hayden Kale and subsequently to 6809 assembly by Jfed.

Balloon gone WILD - Main code discussion

If you've already run the main program, you might be curious about its inner workings. We covered a lot of ground leading up to this point, and some of these topics are advanced, but let's discuss the core algorithm and the use of `bitmap`.

The program starts by defining variables. Lines 120-130 paint the screen with stars using three commands not yet discussed. You can probably guess what each does:

```
bitmap on [off]
bitmap clear {color#}
plot color {color#} to {x pos}, {y pos}
```

screen and pixel color from the palette we have been working with

x and *y* of pixel

Using `bitmap on` (or `off`) is similar to use of the `sprites` keyword. It activates a 'layer' and initializes the subsystem. The bitmap layer occupies the rear layer, sprites are closer forward (towards the user), and text is closer, still; in front.

This use of `plot` on line 130 chooses a random color from 1 to 128 and then plots a pixel from 0..319 on the *x* axis and 0..239 on the *y* axis. The color palette used is the very same palette that we have been working with in our sprite work.

A nice side effect of this is (to the extent that you are lucky), you may have one or more plotted pixels that appear to twinkle (change color) in the sky. This will occur when the random number code chooses color #2, 3, 4, 8, or 9:

- recall, color 2, 3, and 4 are used for flame animation and will cycle: yellow, orange, and red during balloon ascent
- color 8 (the rapidly changing "W!") will change during descent
- Color 9 will alternate from green to gray (the beacon)

If, upon running, all stars appear static, break the program and restart. Of course you can increase the likelihood by increasing the # of 'stars' rendered (currently 256) or limiting

the range of the random number in the color selection portion to a lower number (but too low will make it boring!)

```
for n = 1 to 256
  plot color random(128)+ 1 to random(320), random(240)
next
```

(256 stars)

(Color # to choose from; 1..128)

Three things to try:

- limit the # of colors to a low number; you will be far more likely to have a large number of twinkling stars, however, the variation of color will be far less interesting. Also, during ascent, many of the rendered stars will fade to blue (this is an artifact of choosing to blue-out the flame rather than modify a dozen bytes of the sprite image, which would be expensive).
- change the # of stars generated to 1000. You will be all but guaranteed to have a noticeable quantity of twinkling stars; you may also notice a large majority of the non-twinkling stars are in the blue/green color range. This is due to the lackluster default color palette (see the bottom of pg. 52).
- Try loading up *spredit*, then quitting it (with CTRL-C). Don't forget to type `cursor on`; then reload balloon setup and run the main again. See the difference? This is due to the better distribution of colors in *spredit*'s palette (see top of pg. 53).

Main loop

The original Commodore 64 balloon used a simple diagonal path with no variation. Our WILD balloon is far more interesting. It bounces off walls and simulates powered flight during ascent, chugging uphill. Then, upon reaching the top of screen, the engine shuts down and gravity takes hold.

As downward velocity increases, stability becomes erratic (seeded with a `rnd(0)` calculation). Finally, the balloon lands, pauses, and cycles upwards again.

All of the action is crammed into 15 lines of SuperBASIC code, organized in an endless loop; `goto` statements are used liberally, as we will discuss.

The loop does two big things; first, it propels the balloon upward at a 45 degree angle until it reaches a screen edge; defined on line 100. If it hits the `right` or `left` edge of the screen, it changes direction (tracked with `heading`). But when it hits the `top` or bottom (`bot`), it toggles between ascend and descend (the `ascent` variable tracks this state).

The second big thing it does is animate the sprite by manipulating the color palette three different ways:

a) the beacon at the top of the balloon flashes from green to a gray any time the `x` position of the balloon is in motion (or unless it has landed); b) the "**W!**" flashes with a few dozen hues, randomly chosen from a primarily color scheme (while descending only); and finally, c) there is a simulated flame that rotates its color from yellow to orange to red, and back again, while in powered flight towards the top of the screen.

There is a actually a third big thing going on; simulated gravity while drifting in unpowered flight. While descending, an increasing amount of instability occurs.

All of this action is managed through two integer variables, three state variables, and one floating point, as follows:

`x`	tracks horizontal position from `left` to `right` and is always in motion, while not landed
`y`	tracks vertical position from `top` to `bot` and is always in motion, while not landed
`gravity#`	can be thought of as a gravity *delay* between balloon moves. It starts with a value of 300 while landed or in ascent and peels off 1.4 each time `y` is decremented. NOTE this variable is a floating point number (the '`#`' notation)
`ascent`	1 = up, 0 = down
`heading`	1 = leftward, 0 = rightward
`beacon`	a binary flip/flop variable to manage state

What about multi-tasking

SuperBASIC is powerful, but it is a simple, interpreted language and thus, it has no threads or task management. Further, there is no easy way to orchestrate CPU based interrupts (commonly known as IRQ interrupts on 8-bit processors); so how does this program achieve all that it does, relatively smoothly?

The answer: all calculations, color rotation, and decisioning is based purely on *x* and *y* variables which change in linear fashion (most of the time). Along the way, state variables and simple modulus tests are used to determine when to call a `proc` (such as the beacon flash); and there are cycles to spare.

Commercial games use a update technique called “frame pump”, which is tied to a hardware clock (per refresh of frame). Between updates, *work* can be performed such as checking for input or updating values. As long as the ‘work’ is complete before the next clock tick, game play will feel smooth.

In many BASIC languages, it is common to *burn* time using a simple `for`/`next` loop which does nothing between, such as:

`for x = 1 to 12000: next` (we do this on line 300)

But what if your machine can be configured to run at different clock rates (6.29 MHz vs. 12.6 MHz)? To accommodate, we use the SuperBASIC keyword `timer()`, which returns a counter incremented by the 60 or 70 Hz frame refresh between loop iterations. By capturing the counter ‘now’ in a variable, we can loop until a calculated number of SOF updates have elapsed. There is a peculiarity with this method since it resets from its highest value back to zero every several days, but this is preferred over the ‘do-nothing’ `for` loop noted above.

Difference between `goto` and an all-`proc` based approach

Structured code, such as Python, Java, or C++ (or a vintage hobby based BASIC language that supports procedural programming), is easier to read, write, and debug than the flow control methods that originated with BASIC’s first release (see

Dartmouth BASIC). Even the early versions of Microsoft BASIC, up through the mid 1980s, relied solely on `goto` and `gosub`. Acorn, (via BBC BASIC) on the other hand, supported `DEF` and `ENDPROC` keywords as early as 1980 and thus, SuperBASIC has gifted us `proc`/`endproc`.

As beginners, we were all introduced to the topic of flow control using `goto`, and in some cases `gosub`/`return`. The very first two-line example (pgs. 12-13) is `goto` based. As a flow control method, it is easy to convey.

To close this chapter, we'll examine two versions of the *Balloon gone WILD* program. It is not a strict apples-to-apples comparison, however, the `goto` version uses some of the same procedures (but has a main loop block packed with `goto` statements).

The `proc` version, on the other hand, has an added feature (retro-fire during descent), but you'll note the similarities and differences if you compare the two side-by-side.

To support the premise of a single screen program (in the `goto` version), we had to take a shortcut. In both, we move the graphic definition and load code into the setup program (this is discussed on pg. 54). In the `goto` version, we squeeze more than a single statement per line. In general, this is not a good practice; it's an artifact of an old school style. Download any PET BASIC program or even published programs in computer magazines and you'll see this style.

Core of the main loop of the `goto` version

Two decision statements (on **line 150 and 160**) use the modulus operator '`%`' to mask the balloon's *x* position against 20 (for beacon flash) and 4 (to cycle the engine). Note that in the latter case, the balloon must also be in descent.

Since we are calling procedures, control is returned to the calling line which allows the flow to continue normally. The `gosub`/`return` branch method is similar, but still relies on line numbers.

Line 170 is the screen position update, and is the only place where the `sprite` keyword is used across the entire program.

Line 180 packs a few keywords and a somewhat intimidating expression on a single line. If you look closely, you will see an innocent looking `for`/`next` loop followed by a `while`/`wend` loop. We will discuss this below, but in short, the purpose of this line is to invoke a delay between loop iterations; the algorithm is driven by the `gravity#` floating point variable.

Line 190 through 290 controls the *x* / *y* update, conditionally directing the program flow or altering a series of state changes which test if:

- in ascent
- at top or bottom
- has landed
- heading 'out' or 'in'
- at left or right screen edge

While descending, calculations are performed to subtract from the gravity delay value (1.4) during each loop iteration. This increases descent speed by shortening the delay.

Depending on a combination of `gravity#` and `rnd(0)` at **line 220**, program flow can jump to the top of the loop, creating erratic and turbulent behavior. Of course, this is all an illusion; we are merely using randomness to skip the x-update (or not).

A helpful byproduct of this is a momentary increase in rate of fall since a few lines of code are bypassed. This furthers the perceived out-of-control behavior.

This is what programming is BASIC is all about. Using inexpensive methods to add features without a lot of code, evaluations, or complexity. Keywords such as `sprite` perform dozens of updates and test conditionals behind the scenes.

There is more to this routine as you'll see if you study the code, but the intent is to demonstrate what can be done with just a few handfuls of code, leveraging some modern programming techniques and SuperBASICs power.

A quick walk-through of 5 shared procedures

The *goto* version `proc` definitions also appear in the *all-proc* version. They define or redefine the color palette and one manipulates registers directly without using the `palette` keyword. It's worth discussing in detail.

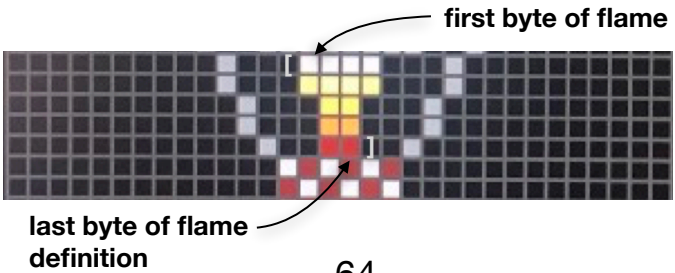
`define_palette()` - is only called once and does what it says, it defines the first 10 palette entries. (the 10th is used for `bitmap clear` to paint the background dark blue).

`eng_on()` and `eng_off()` - overwrite entries for color number 2, 3, and 4 (see table on pg. 65). They either initialize the starting colors for red, orange, and yellow or (in the case of 'off') turn them all dark blue during descent. Yes, we are using the background color as a substitute for transparency.

It's tempting to think that zero'ing these palette entries will make them transparent (more desirable than dark blue), but there is a nuance to consider: `palette` changes the color *definition* for RGB values. The value 0 turns the brightness of a color off, it doesn't make it transparent or remove it.

To make the flame section of the balloon truly transparent, we need to write zeros (the transparent 'color' number) into the bitmap definition in groups, using `memcopy` (to upper memory as we did in the setup program). We might use 5 `memcopy` statements (one per scan line, copying 4 or 2 bytes, each). This could be done in a single, linear run, but we would have to add logic to be sure we were not overwriting the 'ropes'.

`flash_beacon()` - based on the state of the `beacon` variable, will change palette entry #8 to green or gray, flipping the state to animate the light at the top of the balloon.



`cycle_eng()` - called each 4 horizontal pixels, `poke` is used to select I/O bank 1 and alter the $d000-$d3ff address space.

This works in conjunction with variable `eng_state`, rotating its value from 1 to 3. When incremented to 4, it resets to 1. Since none of the flame colors use the 'B' (blue bytes), only 3 `poke` byte-writes per step (on line 1030, 1060, 1090) are needed.

This is more efficient than having to perform 3 full palette commands per step. How much more efficient? It's negligible, at this scale (but 43% faster during a 7.5 second test). We did it this way to highlight an alternative method*

As you transition from working with SuperBASIC to Assembly Language, you will need to learn how to manipulate registers directly. Consider this a stepping stone to achieve that goal.

Here is the complete color palette definition:

Palette color	RED	GREEN	BLUE	Balloon use
0 (transparent)	n/a	n/a	n/a	n/a
1	123	0	0	basket 'a'
2	255	0	0	flame 'a'
3	255	128	0	flame 'b'
4	255	255	0	flame 'c'
5	128	128	128	balloon outline
6	160	160	160	balloon spots
7	180	180	180	balloon body
8	0	255	0	beacon
9	0	0	0	letter 'W!"
10	0	64	128	bitmap

* see pg. 74 for an advanced discussion on SuperBASIC's Sprite Engine, "Five ways SuperBASIC makes your life easier"

All `proc` based version - more code, but is it really better?

If you study the two versions side-by-side, you'll notice several differences. For starters, the procedure based version has three additional `proc` blocks. We won't spend too much time on them here, but briefly, the first replaces line 12000, which is the code executed when the balloon has landed; it is aptly called `landed()`.

Second, you'll see there is a procedure called `move_x()` but there is no `move_y()`. This part is difficult to explain, but the short version of the story is, sometimes a refactor of code is best tackled by removing sections that are called from more than one place. There are other cases where odd logic paths are forged by hardwiring `goto` statements and the remedy is to either reverse the logic or to pull out blocks of code that are relocatable, and that is what was done here.

Finally, there is the `wait(ticks)` procedure, the basis of which is discussed at the bottom of pg. 61. It leverages a hardware timer and a `while`/`wend` loop to reliably spin until the requested number of ticks have elapsed. This procedure will be more meaningful once the 12 MHz core is available because it will allow code to run consistently, between the two clock speed machines. For now, just know that this method is preferred as compared to a `for`/`next` delay loop.

Let's talk about the main loop

Wrapped within an endless "`while 1`", the main portion of the program will loop forever (until CTRL-C) in the same way that a `goto` will unconditionally transfer program control to a hard coded line number. In this case, however, since `1` is an immutable constant that is non-zero, the matched `wend` causes program control will revert back to the start.

It's much easier to visualize program flow when listed, and you can thank `list` *command nesting* for this. There is, however, one peculiar corner-case to discuss.

Consider the highlighted `else` below:

On the left, it appears that if `ascent` does *not* = `1` then the code will do nothing. The reader needs to eyeball the hidden `else` noted at the red diamond which is formatted correctly on the right.

decent (but misleading)

```
if ascent = 1
  y = y - 1
  if y = top_of_screen
    ascent = 0
  endif
  else ♦
  y = y + 1
  if y = bot_of_screen
    landed()
    else
    wind = int((rnd(0)*5)*gravity#)
  endif
endif
```

VS.

much better

```
if ascent = 1
  y = y - 1
  if y = top_of_screen
    ascent = 0
  endif
else
  y = y + 1
  if y = bot_of screen
    landed()
  else
    wind = int((rnd(0)*5)*gravity#)
  endif
endif
```

Both versions run properly; the issue concerns `list` command formatting, not execution. This issue has been fixed in a beta version of SuperBASIC and may have been incorporated into a released build by the time you read this. If your K2 is Wildbits branded, it shipped with this enhancement.

Squeezing in one more feature…

Of course, there is one added feature thrown in and it fires retro-rockets during descent; if you watch closely, you'll see that there is some randomness injected here as well; `wind` and `brake_cntr` variables track movement as follows:

If `gravity#` multiplied by 5 plus a random value is greater than 50, `move_x()` is executed as normal (a 1:1 x-to-y move ratio).

But if not, a variable `brake_cntr` is incremented and when 5 is reached, engines are 'ignited', but only gently; to reduce the rate of descent. This is accomplished by boosting the `gravity#` value by 150 which creates more delay between cycles.

If you are curious, a value of 150 is usually burned off in about 11 vertical pixels worth of normal descent (1.4 x 11).

Note that this is not a one-time action. Each iteration, once `brake_cntr > 5` (conditions meet) will cause the engine to pulse again. Watch closely across multiple round trips and you'll see how this works.

Mimicking the physical world - capabilities and reality

There was no application of physics principles or gravitational calculus used in creating this demo; in fact, attempting to use complex floating point math will most certainly be a drag on performance, leaving less (or no) time to manage graphics or audio (and there is no fun in that).

Calculations on mass, inertia, and even simple trig functions requires a combination of table lookups and computations on the fly. While SuperBASIC does not yet offer `sin` , `cos`, or `tan` and other common functions (present in Dartmouth or Microsoft builds of BASIC), it has other strengths and talents.

Your K2 has an in-line assembler (derived from BBC BASIC) incorporated within. Assembly language is your CPUs native language, and while challenging to master, a lot of work can be offloaded with a little code and a modest amount of effort.

In addition to the FPGA based graphics subsystem and the SID and PSG emulation mentioned above, there exists a set of coprocessor *blocks* to accelerate math functions such as 16-bit division, line-draw, and more. And many of these features will soon be incorporated into SuperBASIC.

The K2 also has a memory manager which expands the footprint of the 16-bit addressing space to reach the upper banks of Flash and RAM. Finally, there is a DMA transfer feature that can move small or large amounts of data without CPU intervention.

Gaining access to control these features require direct memory manipulation via `poke`, `peek`, and `memcopy` commands (discussed above).

If you are interested in this topic, have a look at pgs. 9, 10, and 11 in this Foenix Rising Newsletter, published in 2023. It pits 16-bit FPGA based division against traditional algorithms and illustrates in step-by-step fashion, how to use such a feature.

Of course, since this program is a 'sit back and watch' style demonstration program, all it has to do is move the sprite and calculate the next move. If the delay between frames varies (as it will), the illusion of rate-of-fall is really a matter of delaying *less* between each `sprite` update. We do not need to maintain timing for any other objects or feature such as a player controlled object, or background music or sound.

Ultimately, the performance of a main loop will affect the number of objects that can be kept in motion and still yield acceptable performance. It is your job, as programmer, to leverage a combination of SuperBASIC commands, machine registers, calculations, and pre-calculated data, to entertain the masses. This is the best part of programming.

Some of the best games in history were able to do this in grand fashion with a 1 or 2 MHz CPU and little more than a few player missile style graphics moving around a bitmapped screen.

This is what retro (or what some call *vintage*) computing is all about; doing surprising things with a machine that is resource constrained.

Study formulas and conditionals and then modify them to see how you can alter and improve this program.

Some final things to try:

- modify timing, colors, and the gravity calculation; consider adding a feature that checks for the joystick fire button `joyb(`*x*`)` and if so, 'fire' the engine in a similar way to `wind` and `gravity#` calculations. A goal in doing this might be to touch down at the softest `gravity#` value in a way which is not dissimilar to the ATARI Lunar Lander game. Or, use the new SuperBASIC `at` modifier to `print` a statistic such as:

```
print at 1, 0;"Rate of descent: ";95-gravity#/10
```

- create your own sprite set and animate a rough landing. Changing the image number of the `sprite` keyword is all that is needed to replace an on-screen sprite with a different image. See usage on pg. 38, and the *Sprite Data Set* discussion along with notes on using *spredit.*
- leveraging the spare cycles you have in the gravity delay calculation, consider adding a player controlled sprite that moves horizontally to try and catch the ballon while it is on the ground. This can be as simple as an 8 x 8 animated sprite of a person running. Consult a few YouTube videos of classic Broderbund titles *ChopLifter* or *LodeRunner* for inspiration to animate a small user controlled human.

Now, which version do you prefer?

Do you prefer the long-form, but well structured procedural based version that has no dependency on line numbers, or do you prefer the somewhat less-functional, `goto` based version that you can study on a single screen?

The limits of 60 lines of code feel restrictive but are liberating at the same time since they present an opportunity while posing a challenge.

Pages 76 and 77 contain full listings of both versions with callouts. Study them, and create something of your own.

In conclusion

Hopefully, this handful of SuperBASIC examples provided a spark to support your continued enjoyment. Considering the power in the K2, we have just scratched the surface.

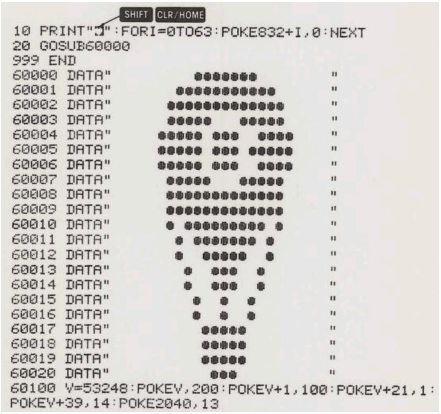
If you've followed the exercises to this point, it is likely that you may have some good ideas of your own. So get your hands on the Wildbits Platform Reference Guide and SuperBASIC documentation, and see what you can create.

Coming soon…

A follow-up to this Getting Started guide will focus on K2 Audio capabilities including the PSG, SID, SAM2965 Dream IC, and General MIDI.

BONUS Material - A quick look at *spredit*

Contained within the Wildbits Graphics Toolkit, *spredit* is similar to image editing tools created in the early-1980s.



Prior the advent of the first editor, the very first sprite example programs used quoted characters to represent pixels; not much different from the type-in examples above. (check archive.org for the original type-in program on the left which appeared in the Commodore Programmer's Reference Guide on pg. 181). Of course, it wasn't long before articles popped up with type-in sprite editors, offering the ability to flip, rotate, mirror, and load or save sprites.

While ASIC technology of the 1980s was innovating rapidly, software was in its infancy; it was years before Deluxe Paint or Real3D for the Amiga were released and considered mature.

The Apple II was first to offer bitmapped graphics on a home computer, but it was the Macintosh that set the standard.

If you were open minded, the original Mac experience was otherworldly, and it didn't take long for others to follow.

Spredit is somewhere between the original Compute magazine sprite editor and a modern tool, but it greatly improves the process of harnessing creativity and does so 'on host'. The value of this should not be overlooked.

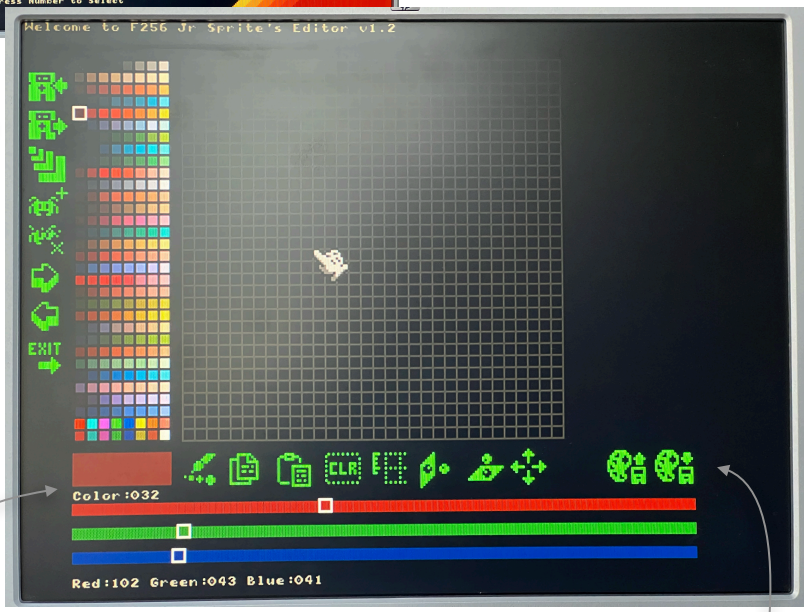
Several things to know about *spredit*

#1 is to read the manual; you can download it here. It contains useful information that you may struggle with, otherwise.

#2 experiment with the on-screen toolbars and clickable regions. You may need to learn some new techniques and will benefit from playing around on a sample project before

depending on it for real work. There is currently no *undo*, and as cited in the manual, there are pitfalls to watch out for such as overwriting files (there is no warning). While it is always good to save copies of your work to different filenames, you can also 'add' a sprite, then copy/paste revisions.

You'll also find that some tools behave differently from modern software, and some are not obvious (such as *color pickup* via spacebar). It's easier to 'pickup' a color than to recall which hue was used in another part of a sprite. If you plan on using palette cycling in your program, this will matter.



Primary tools:
- save
- load
- size
- add
- delete
- next
- prev
- exit

Palette tools

Palette load & save

Useful pieces of data concerning individual colors are displayed on the lower-left portion of the screen. Notice the "Color: 032" (which corresponds with the palette slot number), and the red/green/blue below the sliders.

#3 *spredit* maintains a proprietary library format and it is not load-ready into SuperBASIC. The toolkit includes a *transmute* "Sprites to BASIC" tool, but it adds a step. The spredit manual discusses its use.

#4 remember to save your color palette; this is different from saving sprite data. It is not imperative if you have used the default palette, but if any custom colors were used, you must save the `.pal` (lower right tool) else, they will be gone forever.

#5 Spredit is written in SuperBASIC and leverages sprites (obviously), but also, bitmap graphics, including line draw. Since much of the program is procedure based, it's a great example to learn from and potentially, use in your own programs.

It also uses inline assembly language (not discussed in this guide) and the routines are not very long; so there is a fair amount of useful code that you can learn from.

#6 Spredit is a 2nd generation tool, having been originally developed for the C256U+ platform. If you are interested in the prior version and some interesting tangents, check out pgs. 9-11 in this linked article (clicking will download a .pdf).

A final word about the Wildbits Graphic Toolkit and Tiles

We discussed and worked with redefined graphic characters (see the Wildbits Font Editor), but we haven't discussed the K2's TileMap Engine or the Wildbits Tile Editor (included in the Toolkit and opened via menu item #2).

Tiles came into being in early home video game consoles; the Mattel Intellivision had a 20 x 12 tile playfield and delivered a storage efficiency innovation. Think of a 16 x 16 pixel sprite that can be 'stamped' across a bitmap landscape, requiring only a handful of images that can be reused hundreds of times.

Fields of tiles, organized in a construct known as a 'tile map' can be scrolled smoothly with very little memory or CPU overhead. This YouTube video offers a quick run-through of K2 tiles and their use in an RPG demo staring Valkyrie.

SuperBASIC - Advanced Topics

A few short deep-dives into features and internals.

Sprites - Five ways SuperBASIC makes you life easier

The machine registers within the K2's FPGA contain 8 bytes for each of the 64 sprites as follows:

Offset	R/W	7	6	5	4	3	2	1	0
0	W	—	SIZE		LAYER		LUT		ENABLE
1	W	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
2	W	AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8
3	W	—					AD18	AD17	AD16
4	W	X7	X6	X5	X4	X3	X2	X1	X0
5	W	X15	X14	X13	X12	X11	X10	X9	X8
6	W	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
7	W	Y15	Y14	Y13	Y12	Y11	Y10	Y9	Y8

Comparing what we covered with this list of registers, it's clear to see that SuperBASIC is adding value to make your life as a BASIC programmer, easier.

#1: In the table above, the *x* register is split between byte 4 and byte 5. From early on, we learned that a byte may contain a max value of 255, but the horizontal resolution for graphics is 320 pixels, so how does this work?

The answer is, SuperBASIC manages it all for you. It will toggle the lowest bit of the high-byte when crossing this boundary (while adjusting byte 4) and do the opposite when moving from 256 to 255. If you were writing code in another language such as assembly language, you would need to test for boundaries and manipulate these registers individually.

#2: Bytes 1, 2, and three bits of byte 3 point to the physical address in memory, assuming a contiguous memory map. Above, we discussed the default sprite image location which begins at $03:0000 and explained that the shape is defined along with its size. Based on its internal format, SuperBASIC adjusts these three bytes for you, removing tedium; you need not worry about where in memory sprites live, especially helpful considering that any sprite can be any of four sizes.

#3: Speaking of sprite size, the byte values used in metadata (values 0..3) are different from what is stored in memory. Within byte 0 of each sprite's register set of 8 bytes, bits 5 and 6 represent the size attribute, but with a layer of complexity:

```
small      x11xxxxx : 96   0
medium     x10xxxxx : 64   1
large      x01xxxxx : 32   2
x-large    x00xxxxx : 00   3
```

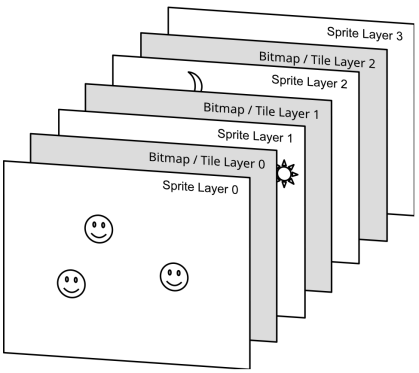
byte mask and value

SuperBASIC size byte value

As a reminder, the SuperBASIC byte value is not defined within your program directly, it's embedded in metadata within the Sprite Data Set (pg. 45). You only have to drop a transmuted blob of data into memory location $30000 (via `memcopy` or `bload`). SuperBASIC worries about this bit business for you.

#4: Layer and LUT complexity is simplified. Graphics capabilities in your K2 allow for multiple color LUTs (look-up-tables) or palettes, and a high degree of layer control. This is useful in allowing objects to pass in front of or behind each other, such as clouds and trees with a fence in the foreground; or to create a parallax effect of tiered mountains in the distance; or to provide a depth of field for near vs. far objects such as a large airplane landing in the foreground as compared to one in the distance, passing behind.

The graphic below shows the K2's layer capabilities from a hardware perspective.



SuperBASIC, however, keeps things simple by providing just 3 layers plus text, as follows:

Tiles reside on the rear-most layer and bitmaps are forward. Sprites are closer still, and text is on top of all. You can change this by cutting out the middle-man and modify the sprite registers (in I/O bank 0) directly.

Let's change the layer priority of sprite #0 to be *behind* the bitmap layer instead of in front of it. NOTE: this is not supported by SuperBASIC but it is a perfectly sane thing to do! Not every sprite register is as forgiving.

We'll use the extra-large sprites from the color palette example on pg. 52 and we will lay them out horizontally, but with a 20 pixel overlap (image #'s are staggered for contrast).

After running the setup program, type the following:

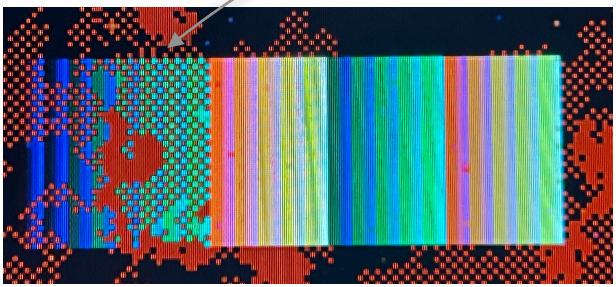
```
sprite 0 image 1 to 100, 100
sprite 1 image 7 to 120, 100
sprite 2 image 2 to 140, 100
sprite 3 image 6 to 160, 100
```

And now, for the magic; type the following:

```
poke 1,0
poke $d900,9
```

sprite #0 is now behind the bitmap

Pretty cool, huh? But be warned, things are about to get weird.



It wasn't mentioned, but sprite priority is based on sprite number regardless of layer, with the lowest numbered sprite, in front.

While it can be desirable to have sprites behind the bitmap (or tile) layer; maybe an automobile driving out of a tunnel (viewed from the side), by doing so, you can force the FPGA to commit *unnatural acts* such as putting sprite 0 behind the bitmap (the code example above), but (here comes the kicker) with sprite 2, 3, and onwards, somehow behind sprite 0 *but in front of the bitmap layer*. To illustrate further, type the following to place an opaque object (a white rectangle) on the bitmap layer; study the callouts on the adjacent page:

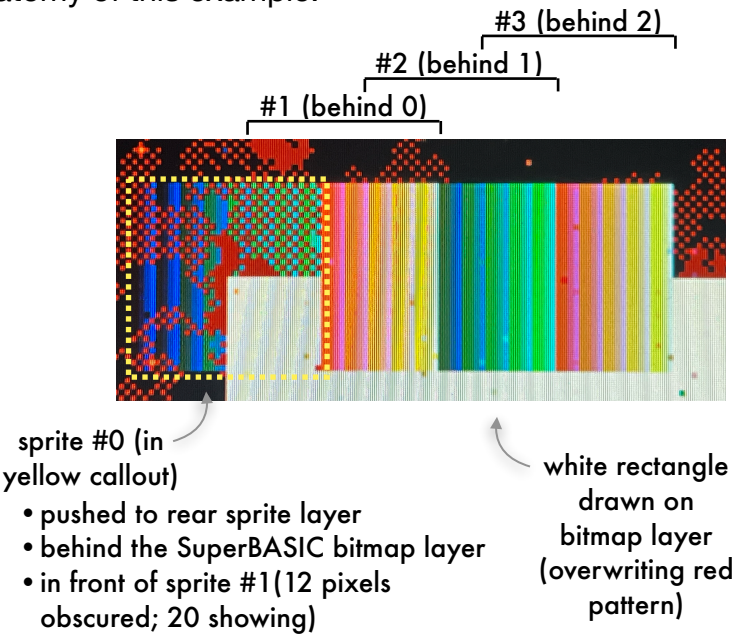
```
rect 100, 100 solid color $ff to 200, 200
```

With the white rectangle drawn (see pic below), you might notice something odd. The red pattern, which lives on the same bitmap layer as the white box, has hundreds of transparent bits, so you can still see portions of the sprite which was pushed to a lower layer with the `poke` on page 74.

But the 'solid' box (no transparent bits), fully obscures a portion of sprite #0 and have broken the planar model in a way, not dissimilar to a mind-bending M.C. Escher drawing.

In summary, SuperBASIC limits functionality by removing some amount of control, but it protects you from odd corner cases that might otherwise leave you puzzled.

Anatomy of this example:



#5: Collision detection with `hit()`

It's expensive, but far less so than doing the math yourself. In the absence of an FPGA based sprite-to-sprite collision detect system, this SuperBASIC keyword is a life saver. Simply:

```
if hit(1,2)              ; test for collision between sprite 1 and 2
  do_explode(2)          ; if non-zero, call a function
endif
```

SuperBASIC - Memory Management Unit and I/O Control

Classically, the 65C02 CPU can address 64K of memory. This was a big deal in 1982, but the glory didn't last long.

Memory Management Unit (MMU) hardware, stepped onto the scene to equip 2nd generation CPUs with bank switching. Constructed within ASICs, they permitted extended memory (up to 2MB) be configured into the 16-bit address space.

Your K2 offers a similar functionality, and does so not only for RAM, but for flash memory, optional cartridge memory, and for the I/O that the machine depends upon to integrate features without being burdensome to the memory map.

Three powerful MMU functions

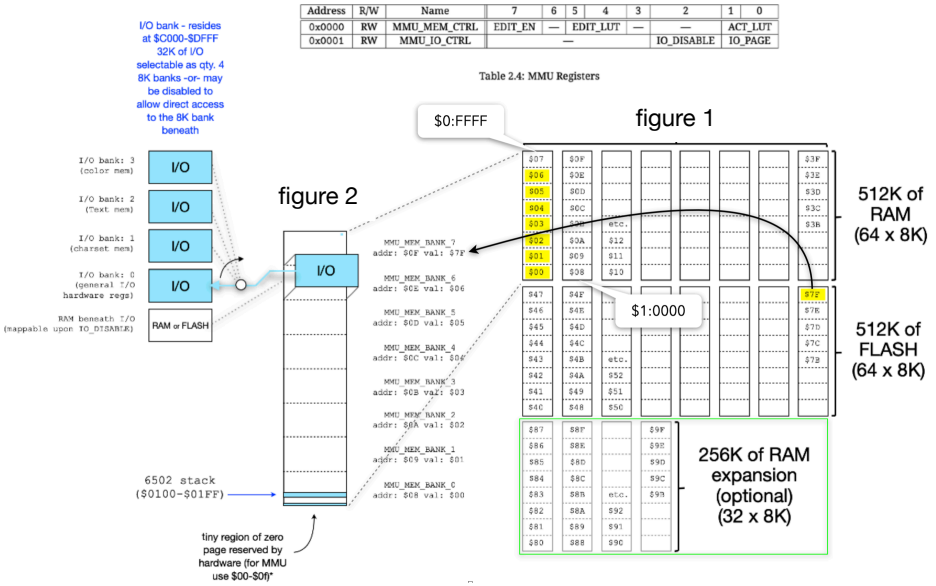
The first, controlled by zero-page address $00, allows selection of one of FOUR user controllable MMU lookup tables (LUTs); these are used to configure 8 banks of memory in nearly any configuration from a pool (fig. 1) of 8K blocks. The K2's MMU was, in fact, modeled after the TRS-80 CoCo Model 3 GIME integrated circuit.

While working in SuperBASIC, this address and the MMU configuration leveraged, should not be altered. There is an intricate *dance* at play between the machine's Kernel (MicroKernel), the FAT file system, interrupt hardware, and the real-time banking of flash (code) and RAM blocks. If you are interested in a bare metal system, however, the full memory map is yours; but you'll need to write your own keyboard, screen, and interrupt and storage interface.

Second, is the MMU I/O Control Register, managed via bits 0..2 of zero page address $01. It dictates which of the four banks input/output features will be banked into the 6th slot at address $C000-$DFFF. See the visual in figure 2 on pg. 79.

It is also possible to leverage the RAM beneath I/O. The `IO_DISABLE` bit (see table 2.4) controls this function. But, similar to byte $00, this memory is reserved for MicroKernel, which SuperBASIC depends upon.

We've already done a fair amount with I/O banks. In example 2 we redefined characters and wrote directly to screen memory and in example 3, we modified the graphics color LUT and above, modified sprite registers. Anytime you see `poke 1,x` you are setting or resetting the I/O system to control banking.



Address	R/W	Name	7	6	5	4	3	2	1	0
0x0000	RW	MMU_MEM_CTRL	EDIT_EN	—	EDIT_LUT		—	—	ACT_LUT	
0x0001	RW	MMU_IO_CTRL	—					IO_DISABLE	IO_PAGE	

Table 2.4: MMU Registers

MMU Performance and a few words about `memcopy`

Requests to reconfigure the memory map (with `poke` or an assembly language `sta $01`) complete instantaneously, but did you know that `memcopy` also uses the DMA subsystem? Consider the following which copies 64K linearly from $01:0000 to $02:0000. How long do you think it will take to run?

```
memcopy $10000, $ffff to $20000
```

The answer: it is near instantaneous. It takes more time to interpret the BASIC code to setup the transfer, than the time required to complete the data move. Clocked with `timer()` before and after the `memcopy`, 2 tics of the 60 Hz clock elapse.

And since `memcopy` is built directly on top of DMA, it can copy rectangle regions of memory, handy for text or graphics objects. Now you are playing with power!