CharAde BASIC Graphics Engine

for the ZX Spectrum, by Jason J Railton

So What Is It?

The **CharAde** engine is some machine code that you can load into a BASIC program to provide faster sprites and scrolling graphics in a BASIC program.

It comes with an editing suite to design graphics characters and blocks. This saves off a block of combined code and graphics which you can load into your own programs.

You only need call this code once at the start of your program, and from then on all commands are accessed through the LPRINT command (except for the odd PEEK where you might need to read back data).

It gives you up to four sets of 64 User-Defined-Graphics (UDGs) with pre-set attributes (that's 256 characters in total) and up to 64 pre-defined 4x4 blocks made up from those characters.

It gives you two hidden over-sized character-mapped screens to prepare graphics on before displaying them on the Spectrum's screen. And one of those screens can be scrolled or used as an even larger map of those 4x4 blocks.

And finally it provides on-demand Keyboard (**QAOP M/N/[SPACE]**) and Kempston Joystick decoding.

Part A - Programming Guide

1. Loading

We'll start with a few quick examples of programming to give you an idea of what's possible, before you go off and start designing your own graphics. An example set of graphics is provided in the file 'EXAMPLE_DATA.TAP'.

If using an emulator:

- First turn off auto-loading of TAP files
- Reset the emulated Spectrum
- Select the file 'EXAMPLE_DATA.TAP'
- Type: **CLEAR 54099** **[ENTER]**
- Type: **LOAD "" CODE** **[ENTER]**
- Type: **LET pk=USR 54100**[ENTER]
- And you're ready to go!

If using a real Spectrum, you'll need to convert the example file to tape, then load it as above.

If using a card storage device like DivIDE, you'll need to do whatever works for you. The 'EXAMPLE_DATA.TAP' file doesn't include a BASIC loader, it's just the CODE block. So you should be able to select it as a .TAP file, then follow the instructions above. At the point you LOAD "" CODE it should load in as if from tape.

And from now on, I'll stop telling you to press **[ENTER]** after every command as it's a bit obvious.

2. The Basics

Let's start with a simple "Hello World" example.

LPRINT "@CØHELLO WORLD)"

If you've done it exactly, beginning with the 'At' symbol, a capital 'C', a zero, capital letters with a space and no extra punctuation, then a close-bracket, you should see the centre part of the screen turn black and the phrase appear in the middle of it. If it didn't happen, check you've loaded it correctly, called **USR 54100**, and typed the command correctly.

So let's break down exactly what is happening. First, when you loaded the example, it loaded in the CharAde engine and then you called it once with **USR 54100** to activate it. After that, everything you send with the **LPRINT** command goes straight to the CharAde engine.

Now in the CharAde engine all the punctuation characters have special properties. So that first '**@**' is like a 'PRINT AT' - it moves the print position (the cursor). The next two characters represent the row ('**C**' is 13) and column ('**Ø**' is 10 - I'll explain later) to move to.

Then the phrase '**HELLO WORLD**' is just ordinary text, so gets printed there. But, this is printed to a hidden screen, so you won't see it happening straight away. So finally the close-bracket '**)**' is a command to dump everything to the actual screen.

Complicated? Ah, there's more.

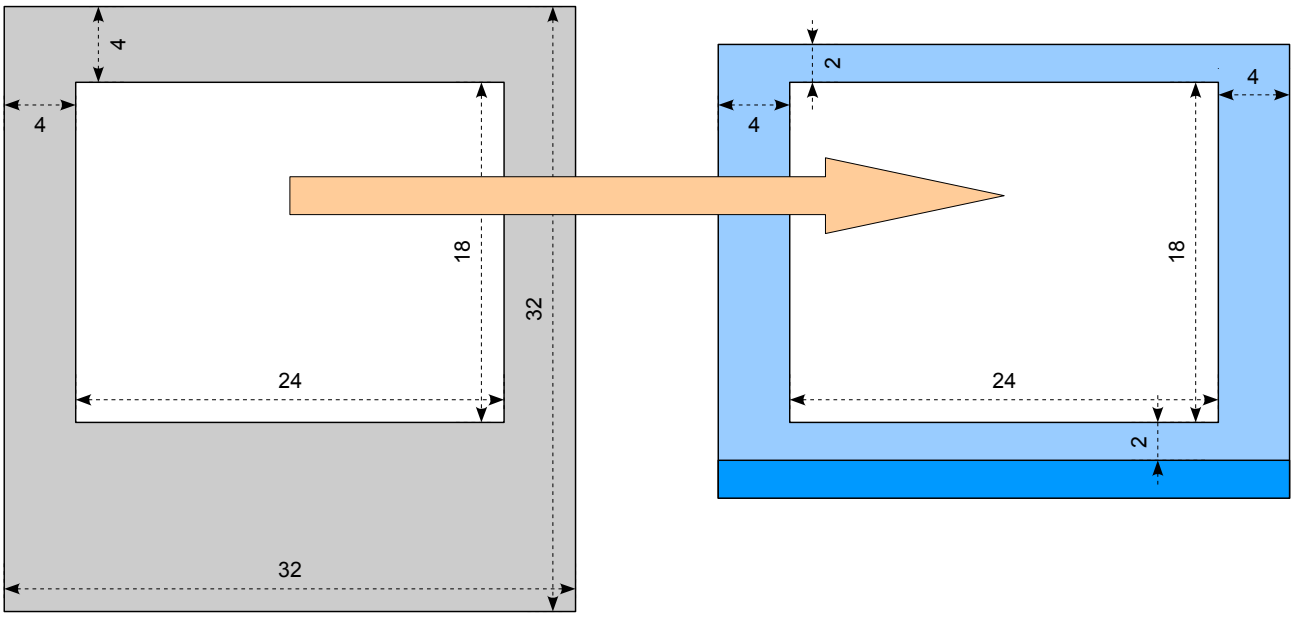
For a start, we were using character set #1 in this example, where the characters actually look like text. Once you start designing your own graphics, those characters could be anything, so trying to print words and phrases will just look silly.

2. Your First Hidden Screen

So, why draw on a hidden screen instead of the visible one? And why doesn't it line up (or hadn't you noticed that)? The main reason is to reduce flicker. In BASIC, if you have to erase something and re-draw it every time it moves, it will flicker. More so if you try to move large graphics or scenery. So, it's smoother if you can build up the picture on a hidden screen, then reveal it all in one go. Also, the hidden screen is bigger than the part you see. This lets you draw large things partially off-screen without running into awkward programming errors.

So, first of all, about the hidden screen. For reasons which will become clear later, I shall refer to it as the 'Foreground' screen. The Foreground screen is 32 characters wide and 32 characters high. If you print to it and run off one side, you wrap around to the other. If you go off the bottom, you come back on at the top. It is therefore impossible for you to get an error from printing 'outside' the screen.

What you may have noticed when the Foreground is copied to the visible screen (type **LPRINT ")"** if you like) is that a window 18 characters high and 24 wide is filled in in the middle of the screen. Two lines are left clear at the top and bottom of the normal printable area, and four columns either side (this is for you to draw a border or scores etc. from BASIC). This window is taken from the foreground screen, starting at a position four columns in from the left, and four rows down from the top. So, if you print something at position (0,0) it will never make it to the visible screen. If you print at position (4,4) then it will appear in the top-left corner of the window when you **LPRINT ")"**.



Transfer of 24x18 window from hidden Foreground screen to actual Spectrum Screen

Now let's do some more tricks to show you why this is worth it. In BASIC, of you want to draw a sprite of 4x4 characters, you need to use an AT statement in your PRINT to move the cursor four times. In CharAde, you only need to move the cursor once, then there's a special 'New-Line' command that takes account of the last cursor placement.

First let's move the cursor to somewhere in the visible window of the foreground:

LPRINT "@55"

Now let's print a 4x4 object - a ring of numbers on the foreground screen:

LPRINT ". 12 . , 8 . . 3 , 7 . . 4 , . 65 . " - pay careful attention to dots and commas

And finally copy it to the visible screen:

LPRINT ")"

See what it does? You can now print large multi-line objects with a single string. You can also store them as single strings for printing later. Perhaps you might store them in an array for animation.

Here's another example - a vertical 'HELLO WORLD':

LPRINT "@5NH W,E O,L R,L L,O D)"

What those commands do specifically:

. - skips over a character space without printing anything, not even a SPACE.

, - move the cursor down one row, and resets it to the horizontal position directly below where the last **@** command put it.

3. An Engine of Character

Now, before I go any further - the purpose of this engine. It's not designed to print numbers and letters, but graphics. If you want to print text or scores, use the ordinary BASIC commands and do so outside the main **CharAde** window. This is about graphics. Use the editor to design up to four separate sets of 64 graphics characters, like UDGs, but each with their own colour attributes. Then get printing them.

Now, with UDGs, you use the characters **A..U** (**A..S** on a 128K Spectrum) in graphics mode to bring them up. With **CharAde**, there's a slightly more complex order to remember.

The first character, representing code 'zero', is the empty **[SPACE]**.

Then come the numbers **1..9**, representing codes 'one' to 'nine'.

Next the bit you will repeatedly get wrong, for which I apologise in advance; the **Ø** or zero on your keyboard is the code for 'ten'. Look, it comes after the 'nine' on the keyboard, right? And zero is already taken, and looks better as an empty space. So **Ø** is 'ten'.

Then come the upper-case characters **A..Z**, counting as codes 11..36, followed by lower-case, **a..z**, which are codes 37..62.

Finally the underscore mark "**_**" represents code 63. Remember we started counting with zero, so that makes sixty-four in total.

Any of these can be redefined as whatever you like. They won't look any different when you type them in a BASIC listing, but they will appear in their new form when you LPRINT them.

Now there are a couple of reasons why you should remember these numbers. Just don't get them confused with ASCII codes, as they're not the same. Most importantly is when moving the cursor with the "@" function. You can use either CHR$() to insert the unprintable characters 0..31, or you can use the equivalent typeable characters from this list.

So you can follow the "@" with CHR$ codes representing row and column:

LPRINT "@" + CHR$(13) + CHR$(10) + "HELLO WORLD)"

Or you can use the letter-equivalents from this list for 13 and 10, which are easier to type:

LPRINT "@CØHELLO WORLD)"

But clearly the first method is simpler if your row and column values are in variables:

LPRINT "@" + CHR$(*row*) + CHR$(*column*) + "HELLO WORLD)"

Note that you can only move the cursor with a row or column value of 0..31. This is equivalent to the first 32 **CharAde** characters **[SPACE] 1..9 Ø A..S**.

And if you do know ASCII, you'll know that CHR$(0)..CHR$(31) are unprintable ASCII characters and stop just short of [SPACE] at CHR$(32) - so the two schemes do not overlap. Feel free to use either.

Just remember that that 'zero' digit represents 10. The empty [SPACE] is actually character 'zero'.

4. The Sets of Four

Yes, there are four sets of characters. You can call them 'Fonts' if you like. Now, you can only work with one font at a time. To switch fonts, use the "**#**" command followed by the number 1..4, or the code CHR$(1)..CHR$(4).

Now remember that I said you could only use one font at a time? The reason for this is that the foreground screen is character-mapped, not bit-mapped. When you print on the Speccy screen it draws 8 bytes that represent the pixels that make up that character. In **CharAde**, it simply places one byte that represents the code of the character printed there - it's a lot quicker, and takes up less memory. This only gets turned into a graphic when you **LPRINT ")"** to dump the Foreground to the visible screen. And it will do so with whichever font is selected at the time of transfer, not when you printed it.

Also, it takes a little bit of time to switch fonts. This is because the dump routine is highly optimised, and mixes the font data in with program code. To use a different font means copying it into all the right places in that code. So don't go switching fonts willy-nilly for trivial animations.

TIP:

For a quick change of scenery in a game, draw different characters to make up walls, ledges, baddies, etc. but line them up in the same places in multiple fonts, so that you can use exactly the same printed objects and code but just switch the font for a different look. Don't be afraid to waste a few characters making your player graphics the same in each font though, so at least one thing is consistent.

TIP:

To start with, stick to just defining graphics with the lower-case letters and leave the numbers and upper-case characters as text you can print. Even when you get more advanced, you may prefer to reserve one font with text characters for title screens etc. You could replace the lower-case characters with little symbols or motifs, or even a few punctuation characters you can't live without.

5. Advanced LPRINTing

Okay, the foreground is probably getting fairly cluttered now, so to clear it use:

LPRINT "'" (That's LPRINT an apostrophe ')

Note that this only clears the hidden screen. You won't see the change on the display until you:

LPRINT ")"

Now some notes about chaining commands together. You may have noticed that following a PRINT command in BASIC, the next PRINT command starts off on the next row. This is because following each PRINT command there is an implicit NEW-LINE character, CHR$(13).

The same happens with LPRINT - after each LPRINT command the Speccy will send a CHR$(13) to the **CharAde** engine, which can confuse it if you're not careful. Usually it ignores it as something that's not a command and that it can't print, but if it's expecting a code, for example following an "**@**" command, it will take that CHR$(13) as the number 13.

So, never do:

LPRINT "@" : LPRINT CHR$(13) : LPRINT CHR$(10)

As what **CharAde** will receive is:

"@" + *CHR$(13)* + CHR$(13) + *CHR$(13)* + CHR$(10) + *CHR$(13)*

And it will act on the first two codes after the "@" and ignore the rest.

You should do:

LPRINT "@" + CHR$(13) + CHR$(10)

So that the "@" command and its parameters are all one string.

You could follow each LPRINT command with a "**;**", like you do in BASIC, to prevent the NEW-LINE occurring. But it's easier to just remember not to split up a single command and its parameters across multiple LPRINT statements.

So, with that lesson in mind, here's your first summary list of printing functions:

' - Clear the foreground screen

@*RC* - Move the print position (cursor) to row *R*, column *C* of the Foreground screen

#*N* - Switch to Font *N* (where *N* = 1..4)

[SPACE] 1..9 Ø A..Z a..z _ - Print a graphic at the cursor position

. - Skip one space without printing anything

, - Move to the next row, directly below the last @ command

) - Copy the window from the Foreground screen to the visible Spectrum screen

6. A Brief Background

So, all this talk of the Foreground screen has to have you wondering, is there a Background screen? and the answer is - YES.

It's another 32x32 character-mapped screen. But what is it good for?

Well, the basic idea is that you draw your game scenery on it once, and leave it alone. You then treat the foreground screen as your 'work area'.

So whenever you want to update your game, you first copy the background onto the foreground (wiping everything that was there from last time), then you LPRINT on it your game characters (sprites), then you copy that lot to the visible screen.

So for that to happen, you need some new symbol commands, and they are:

&*RC* - The equivalent of "@", but moves the cursor to a printing position on the Background screen

(- Copies the background screen to the foreground

! - Clears both the Foreground and the Background screens

Now, you see, you start your display cycle with '**(**', to copy in from the Background screen, then end it with '**)**', to copy to the display - clever, huh? Not entirely mad, now, is it?

Just remember that, like the foreground, the background is 32x32 characters, and the only visible bit is that window 18 rows by 24 columns starting four in from the top-left.

If you want to, have a try now - print some stuff on the Background screen with "**&**". Then use "**()**" to copy it first to the Foreground, then from there to the display.

LPRINT "!#1" - to clear all screens and select Font 1

FOR N=Ø TO 17 : LPRINT "&" + CHR$(4+N) + CHR$(21-N) + "CHARADE" : NEXT N

LPRINT ")" - to convince yourself the Foreground is clear

LPRINT "()" - to copy the Background in and see it

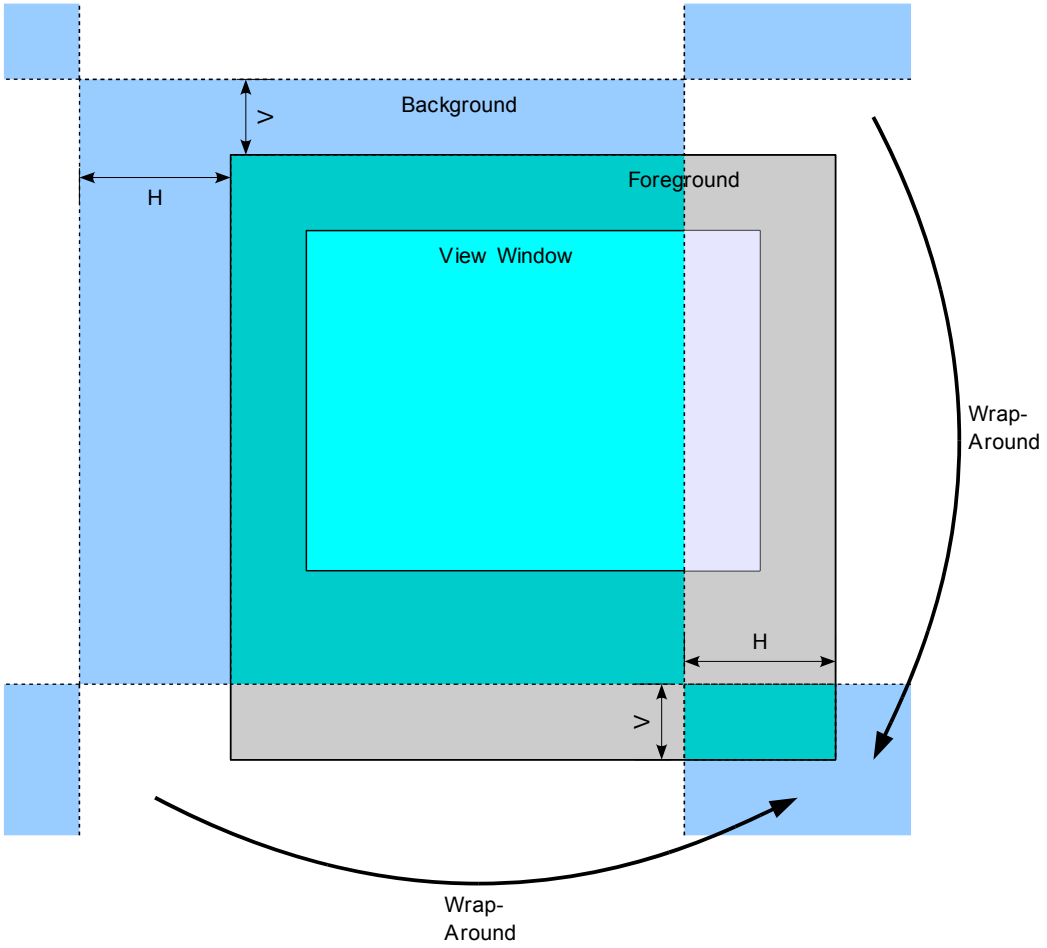
LPRINT ")" - to clear just the Foreground and see it empty again

LPRINT "()" - to see the Background is still available

7. Consult the Scrolls

Okay, now for some real trickery. The foreground and the background are both 32x32 characters. Apart from a bit of border for allowing printing outside the screen areas, what use is all that extra space? Well, how about some scrolling? Here's a new command:

:*VH* - Offset the background and foreground screens vertically and horizontally



Using a Scroll Offset to Shift the Background Relative to the Foreground

Just set up the background again (no need to do this if you just did the last exercise):

```
LPRINT "!#1"
FOR N=Ø TO 17 : LPRINT "&" + CHR$(4+N) + CHR$(21-N) + "CHARADE" : NEXT N
```

Now if you LPRINT "**()**" you'll see the background copied to the foreground and displayed as you originally drew it. But now do:

LPRINT ":␣1" - note the **[SPACE]** then the **1**

And then:

LPRINT "()"

You'll see the background has been copied as before, but shifted one character to the left.

You can keep doing this up to 31 characters either horizontally or vertically, and the background just wraps around when it reaches the edge. In fact, you have to remember to wrap this number around as there are no negative values.

For example, to scroll the screen completely around once to the left:

FOR N=0 TO 31 : LPRINT ":" + CHR$(N) + "()" : NEXT N

Now you have to admit, that's some pretty quick scrolling for BASIC!

To scroll left, the offset goes [SPACE]=0, 1, 2, 3 etc. To scroll right, it goes [SPACE]=0, 31, 30, 29 etc...

Or upward:

FOR N=0 TO 31 : LPRINT ":" + CHR$(N) + "()" : NEXT N

To scroll up, the offset goes [SPACE]=0, 1, 2, 3 etc. To scroll down, it goes [SPACE]=0, 31, 30, 29 etc...

The shift happens when you copy the background to the foreground; anything you draw on the foreground still appears in the same place. For example:

FOR N=0 TO 31 : LPRINT ":" + CHR$(N) + "(@BA,CHARADE,)" : NEXT N

Note that as the screen scrolls, you get to see more of the 32x32 background screen that's outside the 24x18 character display window. You can print things in this area out of sight, and have them scroll into view.

By keeping track of where the right-hand edge of the visible window is on the background (28 characters to the right of the horizontal offset, less 32 if it needs to wrap-around) you can fill in new data just off-screen and have horizontal scrolling scenery that goes on forever.

Similarly, you can fill in just above or below the visible part of the background (remember it wraps around) and give the appearance of scrolling vertically over a much larger area than 32x32.

You can read this data from a map, or you can generate it randomly as you go to create a zig-zagging path, tunnel, or series of obstacles to dodge.

TIP:

Remember to reset the scrolling, you must use **LPRINT ": "** (two [SPACE]s). Doing **LPRINT ":00"** will offset the scrolling by ten characters in each direction!

TIP:

Remember, things drawn on the foreground don't move as the background scrolls. So to make a sprite look like it's moving with the background, remember to subtract the scroll offset from its position. Or just subtract 1 from its position every time you add 1 to the offset.

TIP:

Remember that the viewing window starts four characters down and in from the top-left corner of the foreground. So even when the scrolling offset is reset by **LPRINT ": "** , the background is copied in four pixels up and to the left of the visible part of the screen. To position the background in the top-left corner of the viewing window, you need to set the offset to -4, -4 or 28, 28 with the command:

LPRINT ":RR"

8. Programmer's Block

Now, onto the other thing you can do with the editor - blocks.

Blocks are up to 4x4 chunks made up of characters that you can draw in one go. You can have up to 64 of them (just one set this time), and they appear in whichever font is current. So, if you took my advice and put similar characters in the same place in each of your fonts, you can change the look of your blocks simply by switching font.

Otherwise, blocks designed with one font are going to look a mess in a different font. It's up to you to decide which of your fonts to use with the block designer. You might want to make the blocks work with three fonts for game levels, and use the fourth font for a title screen. Or just one font, as in my examples. Just remember that you can't mix different fonts on-screen at the same time.

Now, you can use blocks for your background scenery, or for your scrolling fill-in function, or for your game sprites. Whichever suits you best.

To draw a block at the current cursor position, there's another command:

$*X* - draw block *X*, where *X* is a character **[SPACE] 1-9 Ø A..Z a..z _**

You use one of the printable characters **[SPACE] 1-9 Ø A..Z a..z _** to say which block, as there are also 64 of them.

Note that in a block, the zero character (SPACE) is never printed. If a block contains a SPACE, then whatever was there before will show through. This allows you to define blocks that only print 2x2 or 3x3 characters, or sprites that have gaps in.

Also, when you print a block, the cursor moves over by two character columns - the minimum you're likely to need. If you are printing 4x4 blocks and you want it to move over the full four, then follow it with "**.,.**". And to move down four places, remember to use "**,,,,**".

TIP:

As well as the empty SPACE character, you may want to define another character as empty and set to your background colour, for use in blocks. Or you may just remember to clear the background before printing scenery blocks on it.

TIP:

Remember that if you print blocks on the background, any SPACEs in the blocks act as transparent. However, when you copy the background to the foreground, everything is copied over, SPACEs and all - the foreground is completely wiped over with no 'transparent' characters showing what was there before.

9. The Big Picture

Okay, one last big feature before we tackle the nitty-gritty. The Map function.

How does this work? Well, when I described scrolling, I said you copy one 32x32 screen to another, with an offset. That's not really all that big, and you have to keep filling-in new scenery just out of sight. What if you could have an area four times as big to scroll around? Well, you can. But it's complicated.

First of all, the map function treats the background screen not as a set of 32x32 characters, but as a set of 32x32 blocks, where each character represents its equivalent block. So, don't print blocks on your background four characters apart; print the equivalent characters next to each other, and the **CharAde** engine will expand them into blocks.

The "**:**" offset function still works as before, but this time offsets the background in whole 4x4 blocks. There's a new command to fine-tune this down to individual characters and draw the map:

;*VH* - Fine-offset a block map and draw it on the foreground

This offsets the map by up to 3 characters in each direction (added on to the whole block offset already defined by "**$**") and draws the part of the block map that covers the foreground. Note that it does not fill the 32x32 foreground screen. It only draws just enough blocks to cover the visible window. More on this later.

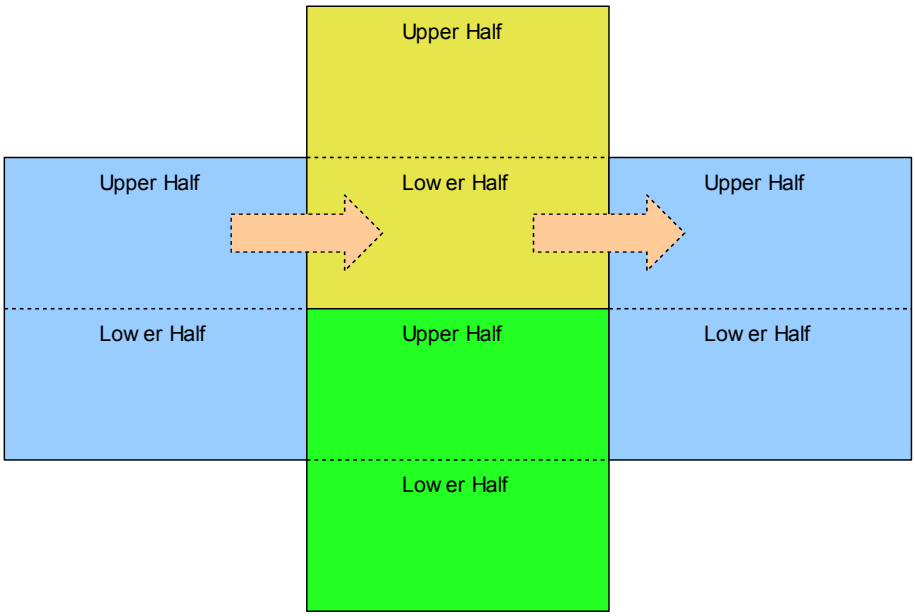
It also follows the rule whereby SPACEs in the blocks are skipped, so show through whatever was there before. You could use this to make layers if you want to - have different layers of scenery in different parts of your map, and draw one over the top of the other. If you don't want gaps, then either make sure your blocks have no actual SPACEs, or remember to clear the foreground before you copy in a chunk of the map.

So, this map now gives you a 128 x 128 background. Is that big enough? Well, it has one last trick. Like the regular background, it wraps-around vertically. Horizontally, however, it wraps-around with a half-map offset. So, if you go off the right of the top half of the map, you come on at the left of the bottom half. And if you go off the right of the bottom half, you come back on the left of the top half.

This lets you treat the map as if it were a 256x64 background instead of 128x128, with the bottom half of the map a continuation to the right of the top half.

Just remember that when actually printing characters (to act as blocks) this special wrap-around doesn't apply. You must still treat the background as a 32x32 grid when printing the characters that represent the blocks.

This special wrap-around only applies when you use the 'draw from the map' function "**:**".



How Multiple Copies of the Block Map Wrap-Around

TIP:

To avoid the wrap-around showing up when you don't want it to, you might want to add an empty border, or a border of solid walls, two or three blocks thick around the edge of the map. That way, if your character reaches the edge of the map, they won't be able to see the wrap-around to the other side. Or, when you get near the edge of the map, stop scrolling, and move your character around the screen instead.

10. Reading From The Screens

There are some ways to read back information from the **CharAde** engine, but you need to let it tell you where to read from. So, capture the result of calling it to initialise it:

LET pk = USR 604100

Now you can use that address for getting information back with a command:

? - Read a character code back from the cursor position

So, to read the character at a cursor position, first move the cursor to the foreground or background with "**@**" or "**&**", then LPRINT the "**?**" command:

LPRINT "@C0?"

Now get the answer to your query with:

PEEK (pk)

Remember that what you'll get back is the code 0..63 for the character, not an ASCII code, nor the character you originally printed.

There are also two commands to scan a small rectangular area, starting at the cursor position and covering a certain height *H* and width *W*:

+*HW* - find the largest non-zero character code within an area

-*HW* - find the smallest non-zero character code within an area

Again, use **PEEK (pk)** to get the result - assuming you've set **pk** with the result of **USR 54100** (okay, the value for **pk** should be **54200**, but be careful, as this may change with updates to the **CharAde** engine).

How useful is this? Well, let's say you're programming a shoot-em-up. You define the player ship and shots in the lower number characters, the enemies in the higher numbers, and the scenery in-between. The background is all SPACEs. Now, to do your collision detection:

i) Copy the scenery into the foreground

ii) Draw the enemies

iii) For each player shot, scan its area for the HIGHEST non-zero character. This indicates if a shot has hit scenery or an enemy and needs to be deleted - but then draw the shot anyway, this time.

iv) For each enemy, scan the area of its sprite for the LOWEST non-zero character. Anything that's not zero but less than enemy and less than scenery must be a player shot drawn over the top of the enemy, so mark that enemy as hit.

v) Draw the enemy shots

vi) Scan the player area for the HIGHEST character. Anything of scenery or higher means a collision with something nasty.

vii) Draw the player ship.

viii) Copy the lot to the visible screen.

If you want to be lazy, or quicker, you can just give the player lasers that pass through everything and don't stop, so skip the testing at stage (iii) and just draw the shots over everything.

And if your player only has one shot on screen, you can skip drawing the enemies at step (ii) and just test the shot for collision with the scenery. Then draw your enemies just after their collision checking at stage (iv). This will be quicker as you can do the scan, then draw them, without having to reset the cursor position. Now you know if an enemy is destroyed, the shot is too.

Note you can read back characters from anywhere on the foreground or background screen, not just in the visible area.

TIP:

Remember how when a map is drawn, only enough is drawn to cover the visible window of the foreground? Well, don't go expecting to be able to read back scenery characters outside the visible area if you're using the map-draw function.

However, remember you can still read back the individual block numbers that make up the map by reading the characters that represent them off the background screen.

11. Control Inputs Made Easy

There are two input functions, which will decode either the keyboard (QAOP and N/M/[SPACE]) or the Kempston joystick, to save you worrying about decoding binary and diagonals:

< - Poll and decode the Keyboard

> - Poll and decode the Kempston Joystick

To get the results, it's a PEEK again. The number read back is a **0** or a **1**:

PEEK (pk+1) = Up

PEEK (pk+2) = Down

PEEK (pk+3) = Left

PEEK (pk+4) = Right

PEEK (pk+5) = Fire

There's also a result at:

PEEK (pk+6) = Tap-Fire

This indicates that the FIRE key/button is newly pressed since last time you asked.

It only occurs once, if FIRE is being held down, then this goes back to **0** the next time you query the controls.

It will only go to **1** again if you poll the controls and Fire is released, then you poll again and it is pressed again.

Note that the **CharAde** engine is not constantly scanning the controls using interrupts or anything. You have to keep polling yourself with **LPRINT "<"** or **LPRINT ">"** to keep this updated.

But it does work independently for keyboard and Kempston, so you can use it in a 2-player scenario.

12. The Last Round-Up

Finally, a few special codes that have escaped our attention so far:

/ - Upward Newline

This works in the same way as printing a "**,**", it just moves the cursor up a line instead of down. So you can print multi-line characters from bottom-to-top instead of top-to-bottom, if you prefer.

******* - Copy background character to foreground at cursor

This just copies one character from background to foreground. This lets you delete a foreground character by replacing it with something from the background. This also takes into account the scrolling offset.

%NRC - Set cursor Row/Column *behind* foreground characters <= *N*

This is a tricky one to grasp but can be very useful. It places the cursor effectively 'behind' the foreground screen. You give it three parameters; mask character *N* (*N* is typically zero, so use a SPACE), Row and Column.

What happens then is when you print a character, it will only be displayed if what's on the Foreground screen is an empty cell (a SPACE itself). If there's already something there, it won't be printed. In this way, whatever you're printing appears *behind* anything that's already been drawn.

If you specify a higher value of *N*, then only character codes equal to or less than *N* get replaced.

So, for example, your first ten characters (**[SPACE] 1..9**) could be various distant background features - empty space, sky, stars, clouds, a path or road, and the rest could be used to draw game characters, buildings, trees etc.

If you set the cursor with **%** and give **N** a value of "**9**" then anything you print will be drawn over those bits of space, sky, path etc., but will go behind everything else.

TIP:

For a semi-overhead view game, where you can go behind some buildings, keep the characters for drawing open spaces and pathways in the lower numbers, and solid scenery in the higher numbers.

First use the "**?**" or "**+**" command at your character's feet to see if the ground they're standing on is obscured by solid scenery. If not, use **@** to draw them over the top of everything (remember you can use "**/**" to draw them from bottom-to-top).

If their feet are obscured, use **%** to draw them behind the solid scenery, but over the top of anything else (should their head or body still stick out somewhere visible).

"*RC* - Set cursor Row/Column on actual display screen

This is a not-so secret command left in from testing that lets you print characters and blocks on the main display screen. That way you can go outside the normal 24x18 viewing window and display your graphics in the surrounding area. The editor uses this feature to display the results of your editing.

Note that you could use ordinary BASIC UDGs for drawing the area around the display window. The editor does this too.

Remember that to type a **"** character you have to enter it twice, for example:

LPRINT """"12HELLO"

And finally:

=*NCFM* - Redefine characters *N*..(x*C*) as font *F* character *M*..

This last function is provided for animation. You may have been tempted to use the multiple fonts for animation, but don't. Switching between them is too slow. Instead, use the "**=**" function. This redefines characters in the current font by copying them from others, or from characters in another font.

Note that the effect is temporary - as soon as you use "**#**" to select any font, it will be restored to its original definition.

Now the four parameters are going to take some explanation:

N - Start with this character

C - This many characters affected, e.g. **1**

F - The font set to copy characters from, **1 2 3 4**

M - The character in that font to start copying from

For example, Font #1 in the example set contains only letters and numbers, so for this exercise we'll re-select that one and clear the screen:

LPRINT "#1!)"

And if we print characters ABC and DEF we just see those letters:

LPRINT "@BDABC,DEF)"

But if we were to grab the six characters that make up the *Spec-Spiker* spaceship image from Font #3 using the command:

LPRINT "=A63a"

All we have to do is re-display the foreground screen again with **LPRINT ")"** and we see the spaceship instead; those six characters have been redefined using images from another font. And a quick re-select of Font #1 with **LPRINT "#1"** will change them back again.

13. Honestly, The End

Did I say 'finally'? I meant, 'penultimately', of course. As there is one very last trick, but this isn't handled by a control code character.

Way back at the beginning, you were told to initialise the **CharAde** engine using the command:

LET pk=USR 54100

You can, if you prefer, use the command:

LET pk=USR 54101

What this does is initialise the engine but with one small difference. Now when you use the) command, instead of a 24x18 character display, you only get a 24x16 character display.

What use is this? Well, it means that you can use the 32x32 Background as two separate background screens (each 32x16 characters), with different scenes in the upper and lower half. Use the scroll offset command to set the vertical offset to either (32-4) = 28 or (16-4) = 12 to bring the particular half of the background level with the top of the viewing window on the Foreground:

LPRINT ":R " **LPRINT ":B "**

Some ideas of what this could be used for:

- Simple two-frame animations in your background - moving conveyor belts, spinning fans, flames, gears, or whirling blades.
- Two states of scenery - light and dark, on and off, triggerable by user actions or lighting effects.
- Perhaps copy the dark half of the Background to the Foreground, then change the scroll offset and use "*" to copy a halo of characters from the light version of the Background around your character. And copy the light version in momentarily to simulate a flash of lightning.
- Design half your background graphics with a 4-pixel offset, then have two frames of scrolling in the two halves, for half-character horizontal scrolling.

14. Control Code Summary:

! - Clear all screens

' - Clear foreground

(- Copy background to foreground

) - Render foreground to screen

#*N* - Load character set *N*

@*YX* - Set cursor Y/X on Foreground screen

%*NYX* - Set cursor Y/X behind foreground characters <= N

&*YX* - Set cursor Y/X on Background Screen

"*YX* - Set cursor Y/X on display screen; *Y*= row, *X*=column

[SPACE] 1..9 Ø A..Z a..z _ - Print a graphic at the cursor position

$*N* - Draw block N, move cursor 2 characters to the right

. - Skip one character place

, - Newline to character below last cursor placement

/ - Newline upward

***** - Copy background character to foreground at cursor

:*YX* - Offset background relative to foreground. *Y*/*X* moves UP/LEFT

;*YX* - Set block sub offset Y/X and paint foreground from map

? - Read character at cursor

+*YX* - Scan Y/X area for highest character code

-*YX* - Scan Y/X area for lowest non-zero character code

=*NCFM* - Redefine characters *N*..(x*C*) as font *F* character *M..*

< - Parse Keyboard QAOPM/N/SPACE

> - Parse Kempston joystick

14. New UDG Character (or Block or Data Value) Codes:

00	=	**[SPACE]**
01..09	=	**1 2 3 4 5 6 7 8 9**
10	=	**Ø**
11..19	=	**A B C D E F G H I**
20..29	=	**J K L M N O P Q R S**
30..31	=	**T U**
32..36	=	**V W X Y Z**
37..39	=	**a b c**
40..49	=	**d e f g h i j k l m**
50..59	=	**n o p q r s t u v w**
60..62	=	**x y z**
63	=	**_**

Part B - Using The Graphics Editor

1. Loading

Load the editor by first ensuring CAPS-LOCK is off, then type:

LOAD ""

It loads in three parts - the BASIC program, and then two blocks of code. It will take a few seconds to initialise, then display the main menu.

2. Editing In An Emulator

You might wonder why I've coded the editor in Sinclair BASIC, when in this day and age you'd expect a PC editing suite. Well, if you follow these suggestions, you'll find this just as fast. I'm using **ZX Spin**, and these notes apply directly to that. In your emulator of choice, you'll have to find the equivalent settings.

- I strongly urge you to turn off auto-tape loading. It's all too easy to lose your work by inserting a virtual 'tape' for saving or loading, only to have the emulator immediately reset. Remember you'll have to manually type **LOAD ""** to start.

- Set the emulated processor speed to 14 MHz (4x normal speed) for a satisfactory response.

- Enable Kempston Mouse support. If you've never done this before, it can be a little disconcerting as the application steals the mouse cursor and stops you moving it outside the emulation window. Remember you can always use you keyboard's **START MENU** key to return the cursor to the desktop, or the **HOME** key in **ZX Spin**. Because your desktop cursor does not line up with the emulated cursor, it's best to select the option to hide it. Note that you won't see a cursor on the editor main menu, only when editing. And you must select mouse control yourself.

- You can use the Main Menu options to save and load data to virtual 'tapes', or you can load and save binary data directly from the emulator menu using the code size displayed. If you do this, always return to the Main Menu before loading or saving binary data. This is because some small changes are made to the data whilst editing, which are then revoked when returning to the menu.

3. Ye Menu

Press "**1**" to change the control type on the edit screens (Keyboard, where the keys are QAOP and M N or [SPACE] to select; Kempston Joystick; or Kempston Mouse).

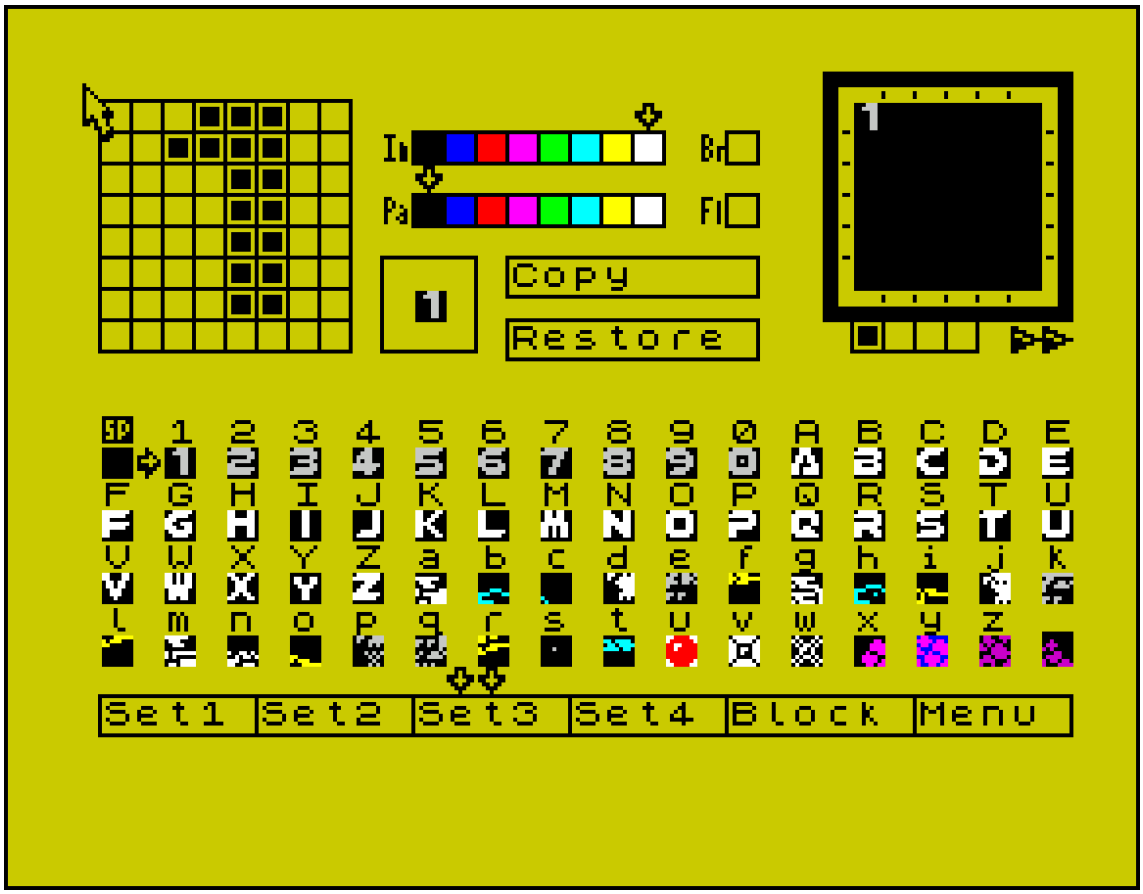
Press "**2**" to swap between editing on a light background (black on yellow) or editing on a dark background (yellow on white).

Press "**L**" to load some previously saved data, or to load the example data file.

Press "**S**" to save the current data to tape. Note that this saves a complete working block of code and graphics, that you can load into your BASIC program as described at the start of the Programming Guide.

Press "**Ø**" to enter the editor. Note that to return to this menu, you'll need to move the cursor to the bottom-right of the editor screen and select the **[Menu]** option.

4. The Character Editor



The 8x8 grid in the top-left of the screen represents the pixels of the current character. Click within the grid to draw or erase. With the mouse, the left button is draw while the right button is erase.

The bars in the top-middle of the screen are used to select the **INK**, **PAPER**, **BRIGHT** and **FLASH** attributes of the current character. Again, click on each box to change. A preview of the current character, in colour, appears to the right of the editing grid.

The centre band of the screen shows all 64 characters in the current font. Click on any one to select it for editing. Under Joystick or Mouse control, you can also press the relevant keyboard key (with or without **[CAPS-SHIFT]**) to select the character. Use **[SYMBOL-SHIFT]** and **9** or **Ø** to select the first and last characters in the font. If you selected keyboard control, you must use **[SYMBOL-SHIFT]** and a letter to select the lower-case characters.

When editing, you can click **[Restore]** to revert any changes back to when you selected the character for editing.

Click **[Copy]** to make a copy of the current character. You'll notice the **[Restore]** option changes to **[Paste]**. You can then select another character and click **[Paste]** to paste the copy.

Note that whilst you have a character in the **[Copy]** buffer, the **[Restore]** function is no longer available. To get it back, click on **[Cancel]** to cancel the character in the copy buffer.

The grid in the top-right is a 6x6 character sketchpad. Click within it to draw the current character. You can use this to draw arrangements of characters for checking sprites, tiles, patterns, etc. Note

that with the mouse, you can also right-click to erase with the SPACE character.

The sketchpad has four separate pages, selectable by the tabs just below it. Also you can click and hold on the double arrows to cycle through the four pages, to test out animations. Note that the number in the top-left of each page is not permanent, and can be drawn over. The contents of the sketchpad is not saved between sessions.

At the bottom of the screen are more options to click on. The first four (e.g. **[Set 1]**) select a different font set for editing. Also, pressing **[SYMBOL-SHIFT]** and a number **1 2 3 4** will switch font.

[Block] switches to the Block Editor, and **[Menu]** returns to the Main Menu.

5. The Block Editor

The Block Editor works much like the sketchpad in the corner of the Character Editor.

At the top of the screen, ten blocks out of 64 are shown. And below those, all 64 characters of the current font. Characters may be selected by clicking or pressing the relevant key (see Character Edit guide).

You can then paint with the character on any of the ten visible blocks. If using a mouse, you can also right-click to erase with the SPACE character.

Beside the blocks which appear at the ends of each row are their number (1-63) and the corresponding character to print with a **$*N*** command, or to print on the background as a map cell. Note that you may not alter the empty Block 0, the equivalent to the SPACE character.

The**[++]** and **[--]** controls on the left move forward and backward five blocks (one row) at a time. The**[+]** and **[-]** controls on the right move forward and backward one block at a time.

The controls at the bottom of the screen select the font set, **[Chars]** returns to the Character Editor, and **[Menu]** returns to the Main Menu.

TIP:

If your blocks come up and look all wrong, think: have you selected the same font set that you designed them with? (The blocks in the example graphics were designed with font set #4).