DaNel.Hogan@schools.pima.gov
Director of The STEMAZing Project

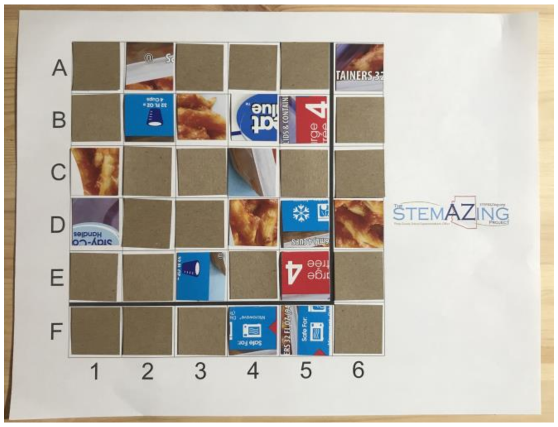
Human Computers – Creative Message Communication

This lesson inspired by CS Unplugged Parity Magic Lesson. (https://bit.ly/ParityMagic)

Turn your students into human computers who can code a message, send it to another human computer who can use parity to check it for errors, correct the errors, and then decode the secret message!

This lesson ideally comes after engaging students with binary numbers using the Binary #MicDropMath Patterns and Build Your Own Cards Lessons found here: http://bit.ly/BinaryMicDropMath You can also find a very rapid STEMAZing PD video about all the lessons in this collection. All resources for this lesson (including presentations in PPT and Google Slides with all the pictures included below) can also be found within the collection linked above under a post with the same name.

After making sense of the binary number system in the Patterns and Build Your Own Cards lessons, students should be engaged using the **Parity magic** lesson from CS Unplugged (https://bit.ly/ParityMagic). This lesson has students use a 5x5 grid of square magnets which are white on one side and black on the other. To make this more accessible and #STEMontheCheap, you can have students cut 1" squares out of a cereal box or other food box which is printed on one side and has cardboard on the other, as shown below. Watching the video on CS Unplugged under the "See teaching this in action" tab or watching the video that goes with this lesson in our collection, you will have a better idea of the computer science concepts behind what seems like a "magic trick" but is actually just math. We are going to continue you on from where the CS Unplugged lesson stops, with parity and error detection, to take it a bit further! Using the cereal box squares, you can actually code a real message! In fact, there is the name of an animal coded in the squares below!

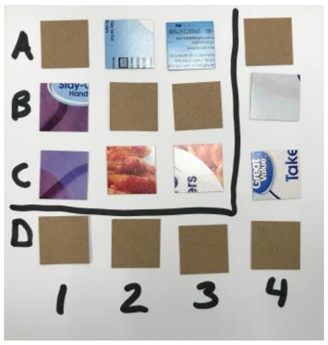


If you are teaching this to students in person, you can do something as a class like they do in the CS Unplugged video with magnets on a white board. Then, students can start to explore on their own by putting 25 of their squares randomly print side up or cardboard side up in the 5x5 grid section with the dark lines around it on their 5-bit parity block. Assuming even parity as they did in the lesson, you can help students add parity bits to the beginning of each row and add parity bits to the bottom row. Elbow partners could check to ensure they have all print and cardboard sides even numbers (0, 2, 4, or 6) in each row and column. Then, one student can turn away from their parity block, another student can flip ONE card over and put it back in the same place, and then the student can turn around and, like a human computer, check the parity to detect the error and then correct it by flipping the card back to the correct setting.

If you are teaching this remotely, you can have students set up their 5x5 grid section with cards randomly print or cardboard side up. You can then tell them how the parity bits should be placed by looking at their setup and instructing them to either put the cards at the beginning of each row, either print side or cardboard side up. Next, you can look away or they can shut off their video and flip one of the cards over. You can then look at the parity block and call out which row and column the error they created is in.

Misconception Alert: A misconception when you are using the 5-bit parity block is that you want to have 3 printed and 3 cardboard sides showing in some rows because they are "even". However, what we mean by even is that the number 2, 4, 6 (and possibly 0 though not technically odd or even but we won't get into that) for the number of either side is even, not that they are equal to each other. So, best to say you don't mean 3 printed sides and 3 cardboard sides, but an even number of cardboard sides (0, 2, 4, or 6) which also forces the number of printed sides to be even numbers (0, 2, 4, or 6).

Differentiation for Younger Students: The Parity and Error Detection lesson can be adapted for younger students by using a 3-bit parity block initially. Students will need to use the 5-bit parity block in order to send messages but starting with the 3-bit parity block will help them master the even parity using smaller quantities. Note that the 3-bit parity block is always in even parity. So, in the 3-bit parity block with the added parity bits, you will always have 2 printed and 2 cardboard sides facing up or all printed or all cardboard sides facing up. Either 2 and 2 or 0 and 4 because it is even parity. Young children should use the 0/1 cards, as shown below, instead of the abstract cereal box cardboard squares.



Differentiation for More Advanced Students: Once students have mastered the 5-bit parity and error detection, you can shift them to the 7-bit system. It is recommended that instead of having students cut out ¾" cardboard squares, you print on cardstock a set of the 0/1 cards they can cut out. These are the cards used in the example shown below.

Even or Odd Parity (It's Your Choice): To confuse this a bit more, you could use odd parity instead of even parity in your parity block. That is, you could agree that having an odd number of each side is the norm – 1, 3, or 5 of the cardboard and printed sides. In that case, you would know there is an error because a card that is flipped over when using odd parity would cause one row and one column to become even. To understand this a bit better, watching this video (https://bit.ly/ParityBitsAndBlocksExplainedVideo) should help. An example of odd parity will be given with the 7-bit parity block. This can be indicated by placing a bead or paperclip on the square next to the parity, even or odd, being used, as seen in example provided.

Human Computers

After you have engaged students with the parity and error detection lesson, explain the 5x5 part of the block of "data" represents information sent from one computer to another. Now, you can let them in on the secret - we can use the cards to actually send a real message! One side of the cards can represent ones (let's say the printed side) and one side of the cards can represent zeroes (let's say the cardboard side). With five zeroes and ones, we can use the American Standard Code for Information Interchange – ASCII (pronounced ask-ee) to send five letter messages back and forth between human computers. And by adding the parity bits to each row and column around your data, just like computers, you can allow the human computer receiving the message to check it for errors before it decodes the message being sent.

Explore ASCII Patterns: But before students send messages, you have to understand the alphabet code – ASCII. Print out the ASCII 7-bit Printable Characters for each student. (We are purposefully starting with the more complicated of the two versions.) Ask students what patterns they notice. This should be alone zone time so they can identify patterns on their own. After you have given students ample time to find patterns, let them share out with the group.

Students should notice:

- The first two digits in each column are the same.
- The last five digits in each row are the same.
- Capital "A" is in the same row as lowercase "a", which means they have the same last five digits. The first two digits determine if the letter is uppercase or lowercase and the same is true for all the alphabet.

Using Binary to Uncover More Patterns: Students who have done the Binary #MicDropMath Patterns and Build Your Own Cards lessons should be able to figure out and write the base 10 equivalent numbers net to each binary number in the 7-bit ASCII. You may need to remind them of the positional notation values for binary for the 7 positions (64 32 16 8 4 2 1). If one of the positions is turned on with a 1, then it should count or be added to get the value of the equivalent base 10 number. Once they have done that, and they should notice a pattern right away which allows them to do this quickly, they should again write down what they notice and what they wonder.

Students should notice and wonder:

- The 7-bit ASCII starts with number 32 and ends with number 127.
- Each column starts 32 values higher than the last.
- What happened to 0-31 – the column that would have started with 00 as the first two digits?
    - According to https://www.ascii-code.com/, character codes 0-31, the first 32 characters in the ASCII-table, are unprintable control codes. They are used to control peripherals such as printers and include things like “Line Feed” and “Carriage Return” and “Horizontal Tab and so on.

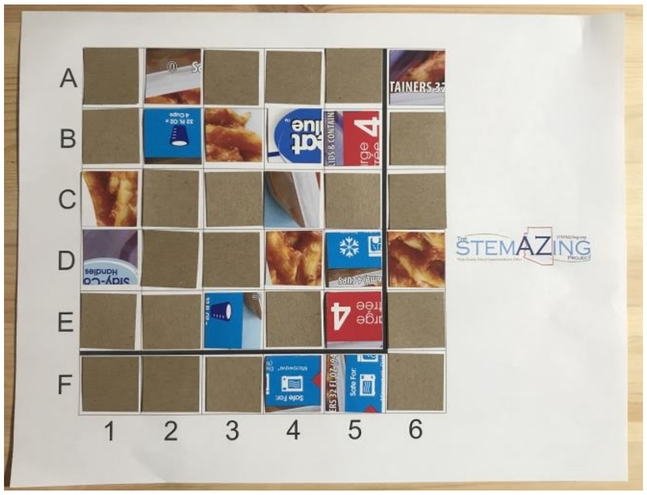
Your Name Out of Numbers (For PCs, does not work on Macs): Another fun trick, if students have access to a PC, is to have them hold down the Alt key and use their number pad to type in one of the regular numbers they calculated next to any one of the ASCII characters. For instance, if they hold the Alt key down and type 98 on the number pad, when they release the Alt key, the letter “b” will appear. If they hold the Alt key, type 42 on the number pad, when they release the Alt key an “*” will appear. And so on through the entire sequence of numbers. You can teach them this by figuring out how to get them to use numbers to spell your name or a fun word. Example: Hold alt, type a number, and then release alt – repeating that for each of these numbers – 68 then 97 then 78 then 101 then 108 – and you have spelled DaNel as intended! There are secret short code codes like this for many other symbols beyond the ASCII printable characters including all the different uppercase and lowercase letters with accents and tildes and so on. The extended ASCII code and more information about this system can be found here: https://www.ascii-code.com/

Keep It Simple To Find More Patterns: Now that students have explored the ASCII 7-bit Printable Characters, give them the 5-bit Space and Alphabet. Ask them to quickly compare this to the 7-bit ASCII Printable Characters. They should immediately notice it is the last five digits for the alphabet characters. Without the first two bits it could represent either uppercase or lowercase which is why both are listed on the 5-bit page. With only 5-bits, you do not have enough space to code for upper or lower case but you can still code for the letter.

Have the students once again figure out the decimal equivalent for each 5-bit binary number. They should quickly see the pattern is that it starts at 0 with the space and then A, a is 1, B,b is 2, C, c is 3, and so on until they get to Z, z which is 26. This means, even if students did not have these cheat sheets for the characters, they could quickly make one again if needed.

You can have students look back at the numbers in the 2nd and 3rd columns on the ASCII 7-bit page. What numbers would they get in the 2nd column if they subtracted 64 from each of them? What numbers would they get in the 3rd column if they subtracted 96 from each of them? Yes – 0-26 just like the ASCII five bit page. So 10----- and 11----- just add 64 and 96 to those columns but in some ways (the last five digits) they are still the same numbers.

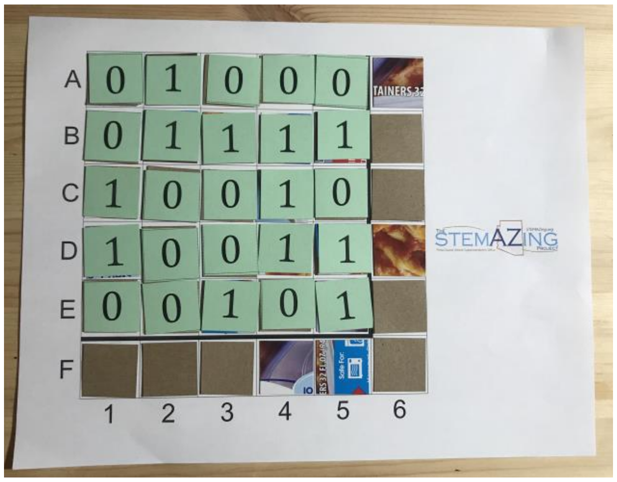
Secret Message Coding: Time to get to the message coding. It is best to have a message coded in your 5x5 grid, which you may have already used to teach about the error detection and correction using parity. An example is given below and it is the same one used earlier in this lesson, there was a message coded there the whole time!



To figure out the message coded in the data above, first one should ensure there are no errors, like any good computer would do. So, check each row and column to be sure there are no odd numbered cardboard sides, which would indicate an error that needs to be corrected. If an error is detected, correct it before moving on. Students also need to know that printed side of the cards is a 1 and cardboard side of the cards is 0. It might be easier for some students to write out the 0s and 1s for the 5x5 data and then decode it. The letters are coded in the rows of data (ignoring the first bit which is a parity bit) so the letters in order are:

```
0 1 0 0 0
0 1 1 1 1
1 0 0 1 0
1 0 0 1 1
0 0 1 0 1
```

Differentiation for Younger Students: You may want to either let them cover the printed and cardboard cards with zeroes and ones as shown below (templates for printing these are at the end of this document) or just have the use 0/1 cards the entire time so it is not so abstract.



Now, it is simple. Students just need to match the codes to the ASCII 5-bit Alphabet to decode the five letter word. And, in this case, you can give them a hint that it is an animal. It is H-O-R-S-E.

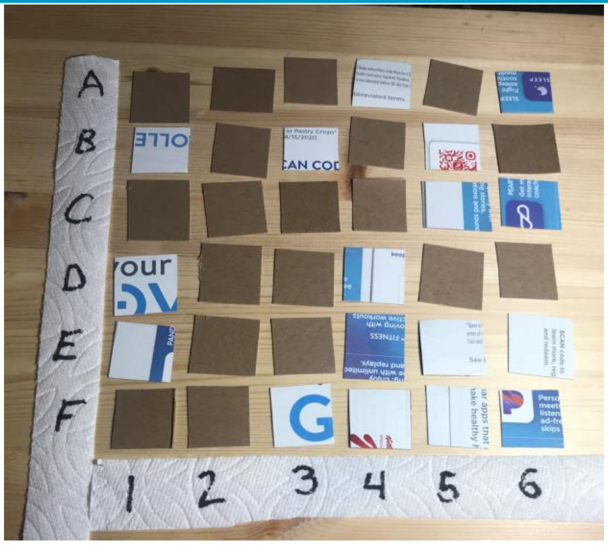
You can have them practice again with the code below.

```
1 0 0 1 1

1 0 1 0 0

0 0 1 0 1

0 0 0 0 1

0 1 1 0 1
```

Did you get it? S-T-E-A-M

One more to practice and without any help. This one is great because you can ask tell students when they think they are done that they should have gotten and animal. If they instead got an article of clothing you wear in the winter, you forgot the first critical step all human computers must do – use parity to detect and correct any errors – and then decode the message. Let them fix it if they got it wrong. The answer is purposely not revealed here.

Here is an example of a coded message that has an error, which must be corrected first. Can you find the error? That's right, the card at B1 should actually be a zero and not a 1. And, if they correctly decode the message, they will find out it says B-E-A-R-S.



Human Computers That Can Code and Decode: Now you can let students code five letter words back and forth to each other. If they want an extra challenge, they can include an error in their message, which must be decoded first. In no case should you let them get away with just coding the 5x5 data set. Make sure they include the error detection and correction parity bits in the first row and last column.

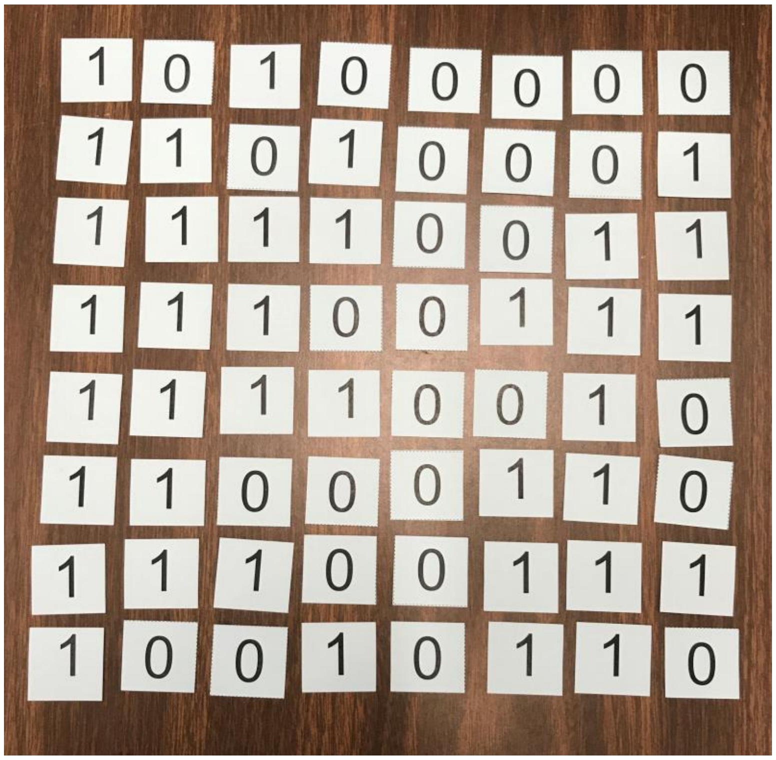
If teaching remotely, you can post coded messages for students to decode and/or they can share their own with the class or each other.

Human Computers – The Next Level

Five-bit ASCII coding is alright but you are fairly limited because you cannot indicate uppercase and lowercase letters. You also cannot indicate any punctuation marks. However, if you step up your coding to a 7-bit data set, you can code seven characters at a time. You still need to add parity bits to the front of each row and a parity byte (an 8-bit column is called a byte) to the bottom of the data for error detection and correction. If you use consecutive parity blocks, you can code longer messages or even entire books if you wanted to. In this case, a message is coded in the two data blocks below. Be sure to check for errors first and then decode. I won't give you the answer to this one but do know that it includes an interesting spelling of one word on purpose. Note, also, that I am using the 0/1 cards to make this a little easier though you could still use the cardboard pieces. Instead of 1" squares, these should be ¾" squares. And, reminding you again, be a good human computer and do not forget to check for errors first!

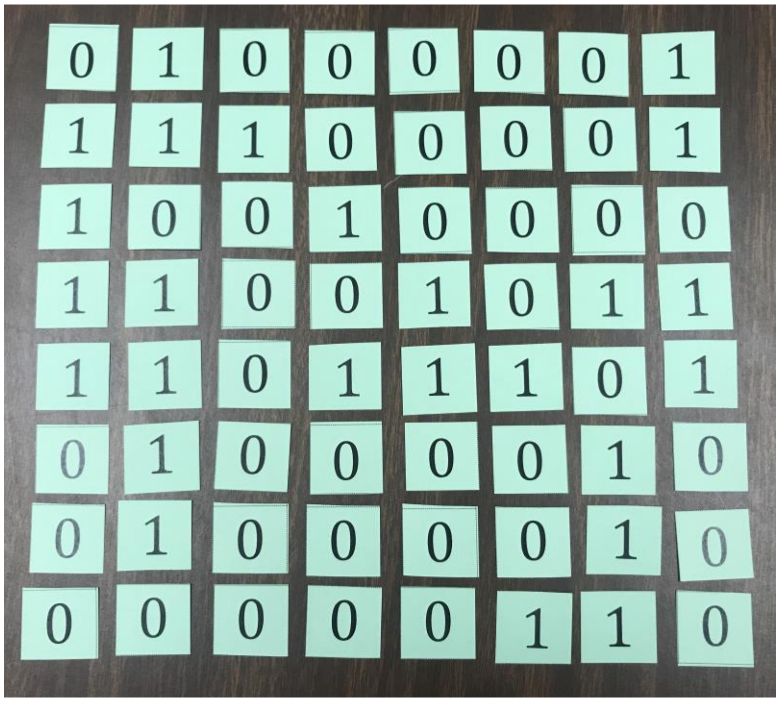
Let the nerdy note sending begin! With a series of parity blocks like those on the next page or using the cardboard pieces or using anything where you can indicate a zero in one case and a one in the other, you can code and send messages using the most common character encoding system in the world – ASCII!

Error Detection, Correction, and 7-bit ASCII Decoding



1	0	1	0	0	0	0	0
1	1	0	1	0	0	0	1
1	1	1	1	0	0	1	1
1	1	1	0	0	1	1	1
1	1	1	1	0	0	1	0
1	1	0	0	0	1	1	0
1	1	1	0	0	1	1	1
1	0	0	1	0	1	1	0

Block 1



0	1	0	0	0	0	0	1
1	1	1	0	0	0	0	1
1	0	0	1	0	0	0	0
1	1	0	0	1	0	1	1
1	1	0	1	1	1	0	1
0	1	0	0	0	0	1	0
0	1	0	0	0	0	1	0
0	0	0	0	0	1	1	0

Block 2

Weak STEAM Extension (but still fun!): ASCII Art and ASCII Animation

Let your students explore the ASCII art here (https://www.asciiart.eu/) and then let them create an original work of ASCII art and/or ASCII animation. There is a lot of great information on the FAQ page here: https://www.asciiart.eu/faq which you might want to assign to students to review before they get started. A few useful FAQs from this site are included below.

Question [2] What is ASCII art?

```
ASCII art is any sort of pictures or diagrams drawn with the
    printable characters in the ASCII character set.
    (For a definition of ASCII, see Question 3.)

    :-) Probably the most common ASCII art picture is the smiley (-:
    =)  but it can get a lot more sophisticated than that.        (=

               ____
            .-" +' "-.      Here's a small ASCII picture of
           /.'.'A_'*`.\     a snow-scene paperweight,
          |:.*'/\-\. ':|    drawn by Joan Stark:
          |:.'.||"|.'*:|
           \:~^~^~^~^:/     If this picture looks very strange and
            /`-....-'\      you can't really tell what it is,
       jgs /          \     don't panic -- see Question 5.
           `-.,____,.-'

    People use ASCII art for a number of reasons. Here are some of them.
    *   It is the most universal computer art form in the world --
        every computer system capable of displaying multi-line text can
        display ASCII art, without needing to have a graphics mode or
        support a particular graphics file format.
    *   An ASCII picture is hundreds of times smaller in file size
        than its GIF or BMP equivalent, while still giving a good idea
        of what something looks like.
    *   It's easy to copy from one file to another (just cut and paste).
    *   It's fun!
```

Question [5] What font do you use for ASCII art?

```
    ASCII art is created using a fixed-width font (like on a traditional
    typewriter), because this is the only way to make it portable.
    However, several Usenet readers now display messages in a
    proportional font (where different characters are different widths).

    The following two lines tell you which kind of font you're using.
    The arrow ends up in a different place for different font types and
    is right most of the time:
```

```
You are using a [Proportional] [Monospaced] font
................................. --^--

Also, to see what your program is doing, look at these two lines:
iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii|
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW|
If they look the same length, you're using a fixed-width font and
all should be ok. If the second line is longer than the first, you
need to change your settings to use a fixed-width font.

Visit site for tips on how to change your font to fixed-width.
```

Question [7] How do I draw my own ASCII art?

Unfortunately, there aren't many text books on the subject. :-)
A good way to learn is to study how someone has made a picture.
What characters are chosen and how the characters are laid out.
How a texture is made.

```
########:::::::::::########
#########:::::::::#########
##########:::::::##########
###########,---.###########
##########/`---'\##########
#########/       \#########
########/         \########
#######:`-._____.-':#######
######::::: ( ) |::::######
#####:::::: ) ( o:::::#####
####::::: .-(_)-. :::::####
###:::::: '=====' ::::::###
########################Mk#
```

The best way to learn is to Practise.
Draw your cat, your toaster, your
partner, your musical instruments,
anything that will sit still long
enough. Practice makes, if not
perfect, then at least pretty good.
Whether you do small drawings (less
work involved) or large ones (easier
to make recognizable) is up to you.
If you're interested in tutorials,
there are many available from the
ASCII-art Documentation Archive.

A good way to begin drawing is to
type a row of spaces for however
wide you want your picture, and
then copy this row and paste it
for however many rows high you
think the picture will get.
Turn Overtype on and place the
cursor somewhere in the middle
and begin drawing. This can save
using Delete, Backspace, Enter
and Space-bar keystrokes.
Saving this empty `canvas' as a
read-only file for future use can
save you even more time later.

```
 _
 \`"-.
  )  _`-.
 ,  : `. \
 : _   '  \
 ; *` _.   `--._
 `-.-'          `-.
   |       `       `.
   :.       .        \
   | \  .   :   .-'   .
   :  )-.;  ;  /      :
   :  ;  | :  :       ;-.
   ; /   : |`-:     _ `- )
,-' /  ,-' ; .-`- .' `--'
`--'   `---' `---' bug
```

Another method is by tracing a picture either onto clear-plastic

```
and sticking it onto the screen then opening an editor to trace
under or using an editor which allows the loading of a background
image to trace over, a process known as `water-mark'.

When drawing ASCII art be aware that there are a few characters
that differ in size, shape and position among fonts:
' apostrophe -- tilts southwest-northeast or vertical
^ caret -- differs in size and shape
~ tilde -- appears in the middle or top
I aye   -- straight line in sans-serif, with strokes in serif
           try using the vertical bar (|) instead.
# hash  -- hash symbol on most, currency on some old computers.
```

Question [18] What is ascii-animation? (If your students really get into asci art, they might like to try this next.)

```
An animated image produced by a sequence of changing ASCII pictures.
The speed will depend on the system you are using.
------------------------------------------------------------------
 o   \ o /  _ o         __|    \ /     |__        o _  \ o /   o
/|\    |     /\   __\o    \o    |    o/    o/__   /\     |    /|\
/ \   / \   | \  /)  |    ( \  /o\  / )    |  (\  / |   / \   / \
------------------------------------------------------------------
Ascii-Animation transports vary a lot. The earliest known portable
types used the Control-Codes of the (often .VT or .ANS) terminal
screens for either `paging' or `direct cursor addressing'.
Sometimes found as c-code in .sigs, which, when compiled and run
produce moving patterns or images.
------------------------------------------------------------------
                  o
                            _        _            _
       _o        /\_       _ \\o      (_)\__/o     (_)
     _< \_      _>(_)    (_)/<_        \_| \     _|/' \/
    (_)>(_)    (_)           (_)      (_)      (_)'  _\o_
------------------------------------------------------------------
Most Web Ascii-Animation uses Java or Javascript.
```

DaNel.Hogan@schools.pima.gov
Director of The STEMAZing Project

ASCII (American Standard Code for Information Interchange) 7-bit Printable Characters

[space]	0100000	______	@	1000000	______	`	1100000	______
!	0100001	______	A	1000001	______	a	1100001	______
“	0100010	______	B	1000010	______	b	1100010	______
#	0100011	______	C	1000011	______	c	1100011	______
$	0100100	______	D	1000100	______	d	1100100	______
%	0100101	______	E	1000101	______	e	1100101	______
&	0100110	______	F	1000110	______	f	1100110	______
‘	0100111	______	G	1000111	______	g	1100111	______
(	0101000	______	H	1001000	______	h	1101000	______
)	0101001	______	I	1001001	______	i	1101001	______
*	0101010	______	J	1001010	______	j	1101010	______
+	0101011	______	K	1001011	______	k	1101011	______
,	0101100	______	L	1001100	______	l	1101100	______
-	0101101	______	M	1001101	______	m	1101101	______
.	0101110	______	N	1001110	______	n	1101110	______
/	0101111	______	O	1001111	______	o	1101111	______
0	0110000	______	P	1010000	______	p	1110000	______
1	0110001	______	Q	1010001	______	q	1110001	______
2	0110010	______	R	1010010	______	r	1110010	______
3	0110011	______	S	1010011	______	s	1110011	______
4	0110100	______	T	1010100	______	t	1110100	______
5	0110101	______	U	1010101	______	u	1110101	______
6	0110110	______	V	1010110	______	v	1110110	______
7	0110111	______	W	1010111	______	w	1110111	______
8	0111000	______	X	1011000	______	x	1111000	______
9	0111001	______	Y	1011001	______	y	1111001	______
:	0111010	______	Z	1011010	______	z	1111010	______
;	0111011	______	[	1011011	______	{	1111011	______
<	0111100	______	\	1011100	______	|	1111100	______
=	0111101	______	]	1011101	______	}	1111101	______
>	0111110	______	^	1011110	______	~	1111110	______
?	0111111	______	_	1011111	______	[delete]	1111111	______

Sign up for The STEMAZing Newsletter!
Follow us on Facebook/Twitter: @TheSTEMAZingPro
www.STEMAZing.org

DaNel.Hogan@schools.pima.gov
Director of The STEMAZing Project

ASCII (American Standard Code for Information Interchange) 5-bit Space and Alphabet

[space]	00000	
A, a	00001	
B, b	00010	
C, c	00011	
D, d	00100	
E, e	00101	
F, f	00110	
G, g	00111	
H, h	01000	
I, i	01001	
J, j	01010	
K, k	01011	
L, l	01100	
M, m	01101	
N, n	01110	
O, o	01111	
P, p	10000	
Q, q	10001	
R, r	10010	
S, s	10011	
T, t	10100	
U, u	10101	
V, v	10110	
W, w	10111	
X, x	11000	
Y, y	11001	
Z, z	11010	

DaNel.Hogan@schools.pima.gov
Director of The STEMAZing Project

Parity Block for 7-bit ASCII Coding

Parity: ☐ even ☐ odd

	1	2	3	4	5	6	7	8
A	🟨	🟩	🟩	🟩	🟩	🟩	🟩	🟩
B	🟨	🟩	🟩	🟩	🟩	🟩	🟩	🟩
C	🟨	🟩	🟩	🟩	🟩	🟩	🟩	🟩
D	🟨	🟩	🟩	🟩	🟩	🟩	🟩	🟩
E	🟨	🟩	🟩	🟩	🟩	🟩	🟩	🟩
F	🟨	🟩	🟩	🟩	🟩	🟩	🟩	🟩
G	🟨	🟩	🟩	🟩	🟩	🟩	🟩	🟩
H	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨

DaNel.Hogan@schools.pima.gov
Director of The STEMAZing Project

Parity Block for 5-bit ASCII Coding

Parity: ☐ even ☐ odd

A						
B						
C						
D						
E						
F						
	1	2	3	4	5	6

DaNel.Hogan@schools.pima.gov
Director of The STEMAZing Project

Parity Block for 3-bit Coding

Parity: [✓ even] odd

	1	2	3	4
A	🟨	🟩	🟩	🟩
B	🟨	🟩	🟩	🟩
C	🟨	🟩	🟩	🟩
D	🟨	🟨	🟨	🟨

0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0

1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

1 1 1 1 1 1

1 1 1 1 1 1

1 1 1 1 1 1

1 1 1 1 1 1

1 1 1 1 1 1

1 1 1 1 1 1

1 1 1 1 1 1

1 1 1 1 1 1