HOOVER-ANDERSON RESEARCH and DESIGN

3x5 Character Set

INTRODUCTION

The 3 x 5 pixel character set is the smallest fully legible character set that can be displayed on a raster-scan CRT. The low resolution of the Arcade screen (160 pixels wide by 88 pixels high) mandates the use of high density characters for text-intensive applications like adventure games, correspondence, word processing and embedded game instructions. The 3 x 5 character set gives the user 14 lines of 40 characters each (560 characters) compared to the 11 lines of 26 characters each (286 characters) available from the on-board 5 x 7 pixel character generator.

The 3 x 5 pixel character set was designed by Craig Anderson for use by the designers of the Z-Grass 32 add-under and was incorporated into that system. When you have coded the following utility you will get a preview of what the Z-Grass 100 screen looks like.

THE PROGRAM

The 3 x 5 character set consists of up to four parts; 1) The Data Base; 2) The Interpreter; 3) the Data Base Decoder and Display Routine; and 4) Text Storage.

1. The Data Base

   Standard ASCII (American Standard Code for Information Interchange) uses 128 numbers to represent all alphanumeric characters, both upper and lower-case, and all control commands necessary to drive any type of printer or terminal. Jay Fenton modified the ASCII code for use in the Bally BASIC and AstroBASIC. He threw out the lower-case codes, since the hardware did not support lower-case, ignored most of the control codes and substituted single-word command statements for some of the lower-case codes.

2206 WEST 21st STREET MINNEAPOLIS, MINNESOTA 55405 612-374-3394

The 3 x 5 character set does not support the single-word commands or lower-case and also ignores all but two of the control codes (ERASE and CARRIAGE RETURN). This freed half of the ASCII codes so we threw them in the bit bucket and reverted to a 64 number subset of ASCII called field data or, in lighter moments, "Half-ASCII".

Most of the punctuation and special symbols remain with the exception of the really impossible and the "who-needs-'em-anyway" such as backward slash, dollar sign, left and right arrow, ampersand, the "at" sign and the crosshatch or "pound" sign.

All but eight of the 64 Half-ASCII codes are simple displacements: subtract 30 from the standard ASCII code and you will get the Half-ASCII value. The remaining eight must be interpreted by the program so we have inserted eight IF statements to check for these values.

The program (Appendix B) shows the 3 x 5 character generator in a typical form: set up for AstroBASIC using the *(n) "reverse string" to store the 64 15-bit values (see Appendix A - 3 x 5 Character Set Data Base - DATA column). To enter this table into the computer, enter the 64 values in the DATA column into the variables *(0) through *(63). The data base, therefore, will require 128 bytes of memory that may not be used for your program. You must stop programming when SZ equals 128.

Suppose you are using Bally BASIC instead of AstroBASIC. Bally BASIC does not support the *(n) variable, only the @(n) "forward string". If you are using Bally BASIC instead of AstroBASIC you must wait until your program is completed, make sure that SZ is at least 128, then enter the 64 cells of tabular data.

Suppose you can't afford the 128 bytes to store the data base in the screen RAM text area. If you are the proud owner of a Blue Ram, a Viper or a Z-Grass add-under, or any expanded memory of your own concoction, you are home free and may store the 128 byte data base anywhere you wish. If not, there are no buffers large enough to hold 128 bytes of data in any of the scratchpad or working areas of the Arcade without its being clobbered by the temporary data that rightfully comes and goes in those areas. You can experiment

with %(n) and store constant strings there (small phrases or words, perhaps) but not the entire character set.

So, if you are working in AstroBASIC, enter the data base in variables *(Ø) through *(63) now.

If you are working in Bally BASIC, the data base will be the last thing to enter and you will put it in @(Ø) through @(63).

2. The Interpreter

As mentioned before, this is a set of eight IF statements that take care of the eight non-displacement values. There is also an IF statement that takes care of any values falling outside of the valid Half-ASCII range.

All nine Interpreter lines are only needed if full ASCII code is being sent. Otherwise, if your text is already encoded in six-bit Half-ASCII instead of seven-bit full ASCII, you can leave out all but lines 18 and 19. The carriage return, line 18, must be modified to read "IF A=61..." instead of "IF A=-17..". We shall cover this more in the section on Text Storage.

3. The Data Base Decoder and Display Routine

This is the heart of the 3 x 5 character set which takes the indexed value from the data base (the contents of strings Ø through 63) and decodes it into a 3 pixel wide by 5 pixel high dot pattern, which it displays on the screen.

The actual Decoder and Display Routine is only 157 bytes long. It, together with the data base, occupies only 285 bytes of memory, leaving 1515 bytes for your program.

Each string position in the data base is a 16 bit word (2 bytes). The bits are numbered Ø through 15, as usual, right to left, with bit Ø equal to 1, bit 1 equal to 2, bit 2 equal to 4, etc. The 16th bit (bit 15) is equal to 32768. The Arcade's Z-80 microprocessor cannot do signed arithmetic if it uses this bit as a numeric value (32768) so this last bit is called "the sign bit", giving the Arcade a range from -32767 to 32767. Since the 3 x 5 character set only needs to set 15 pixels, (3 times 5), it does not use the sign bit. You will notice that all numbers in the data base are positive numbers and that division begins at 16384, not at 32768

or a sign test. Each bit from Ø to 14 therefore represents one pixel. The pixels in the character matrix are arranged as shown in the diagram. The Decoder begins decoding at bit 14 and continues until it reaces bit Ø. You can, therefore, change the data base at will by setting whichever pixels you desire and recalculating the value of the 16 bit word. A black box, measuring 3 by 5 pixels, would have the value 32767.

14	13	12
11	10	9
8	7	6
5	4	3
2	1	Ø

The character set was designed with some human engineering factors in mind. An attempt was made to keep all characters different from all other characters by at least two pixels per character. This was not possible with the characters H, M and W for obvious reasons. All digits have squared corners while all alphabetic characters have rounded corners with the exception of O and Ø, where an attempt was made to follow Bally's 5 x 7 convention of square Os and round Øs. Don't ask why ... we don't know.

HOW TO USE IT

The 3 x 5 character set may be used in two ways: embedded or interpretive.

1. Embedded

Use lines 18 and 19 of the Interpreter, (modified to read "IF A=61 ..." instead of "IF A=-17 ...") and lines 21 through 26 of the Decoder as a subroutine (remember to renumber the lines depending on where you place it). Now you must set the following parameters before your GOSUB instruction:

X = horizontal pixel location of upper left pixel of first character in text (-80 to 76)
Y = vertical pixel location of upper left pixel of first character in text (43 to -39)

Now send your string using

A = Half-ASCII character to display (Ø to 63)

You cannot be currently using the variables A, B, C, D, E, F, X or Y for any other purpose; they will not be returned intact.

Let's suppose you wish to print the words: HI THERE! on the screen beginning at the center of the screen and have stored the data base in a * string. The words: HI THERE! must now be stored. Let's place them in string memory starting at *(64), the first unused string variable.

Enter the following:

```
*(64) = 42  (the letter H)
*(65) = 43  (the letter I)
*(66) =  2  (SPACE)
*(67) = 54  (the letter T)
*(68) = 42  (the letter H)
*(69) = 39  (the letter E)
*(70) = 52  (the letter R)
*(71) = 39  (the letter E)
*(72) =  3  (EXCLAMATION POINT)
```

The program would now be entered as follows:

```
1 CLEAR; X = Ø; Y = Ø
2 FOR G = 64 TO 72; A = *(G); GOSUB 18; NEXT G
3 STOP
```

RUN the above program and you will see the 3 x 5 character generator in operation as the words: HI THERE! appear with the upper left corner of the H in HI centered at Ø, Ø.

This technique is fine for storing short bits of text like HI THERE! For long messages, see the section on Text Storage.

2. Interpretive

Enter the program up through line 26 (you may delete the comment lines 1, 2, 3, 5, 10 and 20). (See program listing: Appendix B.)

RUN the program. It will ask for the beginning X and Y coordinates at which you want the text to appear. Let's say you want to start at the upper left corner of the screen. Enter -8Ø for X and 43 for Y. It will also ask for G; enter any number.

Now type in your message on the keypad. Try the entire character set, especially the control characters ERASE and GO (carriage return).

The program is interpreting ASCII into Half-ASCII. Notice how some characters (the impossible and the "who needs 'em") are either ignored or are interpreted as some other character. The Data Base table will tell you which characters are supported by this set and which are not.

For an interesting demonstration of how the interpretive mode converts ASCII to Half-ASCII and ignores characters it cannot reproduce in a 3 x 5 pixel form, let's do a partial program listing in 3 x 5 characters. Yes, it works. Re-enter the comment lines and add the following lines to the program:

```
110 FOR G = -24576 to -24276 STEP 2
120 H = %(G) ÷ 256; A = RM; GOSUB 11; A = H; GOSUB 11; NEXT G
130 CY = Ø; STOP
```

Delete line 6. Now type RUN, press GO and enter -8Ø for X; 43 for Y and Ø for G. Watch what happens. We'll bet you've never seen that before! We'll also bet you've never seen a program listing like that before either. Where are the command words like CLEAR, IF and FOR? In the bit bucket, as we mentioned earlier. Half-ASCII doesn't support them, so the Interpreter weeds them out and throws them away.

In the above demonstration, the little routine at line 110 is feeding full ASCII codes to the Interpreter which is selecting the codes it can convert to 3 x 5 characters, displaying them, and discarding all others. Among the discards are line numbers which are not decoded fully by the routine at line 110. Where are the ASCII codes coming from? They are coming from the FOR-NEXT loop in the routine above which is PEEKing directly into text memory starting at the first location: %(-24576). That should give you one idea of how to store character strings for the 3 x 5 Decoder. Let's examine some more.

TEXT STORAGE

We said earlier that the 3 x 5 character set consists of four parts. If you are going to use the 3 x 5 character set to display "canned" material such as comments or labels during a program you must have a way of storing that material. Here are some suggestions; there are other ways open to your own imagination.

1. String Variables

The "HI THERE!" demonstration had you storing the Half-ASCII codes as string variables using either the * or @ construct. While it works, it is a very inefficient storage technique. Each "string" variable requires 16 bits (two bytes) of memory to store a 6 bit value (Ø to 63). We do not recommend this technique for other than programs requiring a few short character strings.

2. REM Statements (Interpretive)

You can put longer text strings into REM statements. REM statements are those that are preceded by a period so that they are not processed by the BASIC interpreter cartridge. For example, begin at line 1:

1. I AM A REM STATEMENT. THE BASIC INTERPRETER DOESN'T KNOW I'M HERE.

2. HOW CLEVER OF YOU TO FIND ME.

We must now tell the 3 x 5 Interpreter to ignore the first period (same one the BASIC Interpreter ignores) but to print all periods thereafter. Let's just avoid sending it at all. Notice that the first character after the first period is a SPACE. It takes two characters to fill each memory location, and memory locations progress by twos. For practical purposes (not real purposes; this isn't a tutorial on Arcade memory allocation!) let's pretend that each character occupies one memory location (one byte) but that we are forced to count them by twos, okay? The line number hogs two memory locations. Since it is line number 1 it hogs locations %(-24576) and %(-24575). Let's pretend the period hogs location %(-24574) and the space is in location %(-24573). That means that the "I" of "I AM A REM STATEMENT" is in location %(-24572). It isn't really, but I haven't the time to explain why it isn't and where it really is, and you haven't time to read it. Nor do you care. If you pretend it IS in %(-24572) the program will work as if it really is there. Now, I count 66 characters in line 1, so if we pretend that each character (including spaces and punctuation) takes one memory location, that means that the word "...HERE." ends at memory location %(-24506).

Change line 110 of the previous routine to read:

```
110 FOR G = -24572 TO -24506 STEP 2
```

Type RUN, hit GO, enter X, Y and Ø for G and watch what happens. Pretty neat, huh? It works!

Now, how about line 2? We left off at %(-24506). Let's assume that the line number 2 hogs locations %(-24505) and %(-24504). The period and space occupy %(-24503) and %(-24502). That puts us at %(-24501) for the "H" in "HOW...". There are 29 characters in line 2. Change line 110 to read:

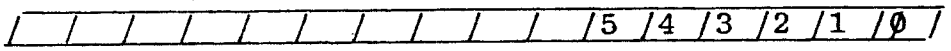
11Ø FOR G = -24501 TO -24472 STEP 2

Type RUN, hit GO, enter X, Y and Ø for G and there you are! Contratulations, you've done it again and you now know a more efficient way to store text strings for the 3 x 5 character generator. How much more efficient? Well, each character now takes only 8 bits (one byte) of memory to store. It's a significant improvement. Remember to insert a SPACE after the first period.

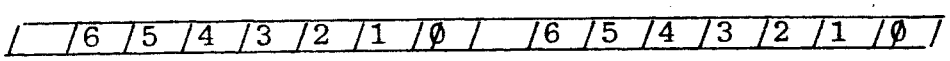
You can stop here if you want to. We didn't, of course. It really kept us awake nights knowing that we were still wasting two bits per byte (25% of memory) and that we could do better still. What we are about to tackle now is the reorganization of the Arcade's memory, a super-sophisticated technique called "high-density packing" that allows us to store more than 1800 characters in 1800 bytes of memory.

3. High Density Packing

The Arcade is presently configured into what we will call "memory cells". Each cell is two bytes in size. You know memory cells by their commands: %, * and @. Storing a Half-ASCII character in a POKE, STAR STRING or EACH STRING looks like this:



We have learned that we can also store Half-ASCII strings in text memory as ASCII strings. ASCII strings require seven bits and must be interpreted. You could bypass the 3 x 5 Interpreter if you could generate the non-printing ASCII commands (Ø through 31) from the Arcade keyboard. Unfortunately, you cannot. Storing text strings as REM statements looks like this:



We are wasting four bits out of the 16. Bit 6 is unneeded and requires the Interpreter program to sort through it. If we scrap bit 6 and write in straight Half-ASCII we are giving the Decoder everything it needs but still wasting 4 bits out of every memory cell of 16. That's 25% waste. It should be obvious that one way to approach this problem is as follows:

```
/5 /4 /3 /2 /1 /Ø /5 /4 /3 /2 /1 /Ø /5 /4 /3 /2 /
```

That's horizontal dense packing. Unfortunately, it ends in the middle of a character. We need three memory cells (six bytes) to come out even:

```
/5 /4 /3 /2 /1 /Ø /5 /4 /3 /2 /1 /Ø /5 /4 /3 /2 /
/1 /Ø /5 /4 /3 /2 /1 /Ø /5 /4 /3 /2 /1 /Ø /5 /4 /
/3 /2 /1 /Ø /5 /4 /3 /2 /1 /Ø /5 /4 /3 /2 /1 /Ø /
```

But look what we've done! We've packed 8 characters into 6 bytes! The Arcade offers 1800 bytes of memory. If we can get 8 for 6 we can store 2400 characters in that 1800 bytes. Wow! Binary Magic! We have just done the impossible: we have displayed more than 1800 characters in less than 1800 bytes!

Well, we haven't done it yet. We must first look at the problem of reconfiguring the memory and weigh some trade-offs. There are two possibilities: horizontal packing and vertical packing. The example last given was of horizontal packing. Each character follows each other character with the character boundaries breaking evenly every three memory cells or six bytes. Here's where the sign bit (remember bit 15, the one we've been ignoring so far?) rears its ugly head. If we use high-density packing we must use the sign bit. The sign bit is not easily read. We cannot, for example, start the value of B in line 21 at 32768, the sign bit's actual value, because the Arcade doesn't have a 32768 ... it has a sign bit. The sign bit must be written by setting a memory cell to a negative, and read by determining whether a memory cell is less than Ø. Writing horizontally would require a lot of calculating to determine which cell we're in of the group of three and when we've changed from one to the next so we can, if necessary, change the previous cell to a negative value. Who needs it? Let's forget horizontal packing and try vertical.

Vertical packing works with groups of six memory cells instead of three and uses twelve bytes. It will store sixteen characters and looks like this:

ADDRESS	BIT NUMBER	VALUE
*(64)	/Ø /Ø /Ø /Ø /Ø /Ø /Ø /Ø /Ø /Ø /Ø /Ø /Ø /Ø /Ø /Ø /	32
*(65)	/1 /1 /1 /1 /1 /1 /1 /1 /1 /1 /1 /1 /1 /1 /1 /1 /	16
*(66)	/2 /2 /2 /2 /2 /2 /2 /2 /2 /2 /2 /2 /2 /2 /2 /2 /	8
*(67)	/3 /3 /3 /3 /3 /3 /3 /3 /3 /3 /3 /3 /3 /3 /3 /3 /	4
*(68)	/4 /4 /4 /4 /4 /4 /4 /4 /4 /4 /4 /4 /4 /4 /4 /4 /	2
*(69)	/5 /5 /5 /5 /5 /5 /5 /5 /5 /5 /5 /5 /5 /5 /5 /5 /	1

±	1	8	4	2	1	5	2	1	6	3	1	8	4	2	1
	6	1	0	1	0	1	5	2	4	2	6				
	3	9	9	4	2	2	6	8							
	8	2	6	8	4										
	4														

The text must be stored in blocks of 16 characters each. If you wish to display more than 16 characters (like an entire letter to your girlfriend of 1500 characters or so!) you must set up a FOR-NEXT loop that starts G at 64 and keeps stepping it by 6 until your text is done, 16 characters at a time.

Delete lines 1, 2, 3, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 20, 100, 110, 120, 130 and 190. (We hope you never bothered entering lines 30 through 90 and 140 through 180; they're just there to make the printer space the program out on the page!)

Change line 18 to read "IF A = 61 ..." instead of "**IF A = -17...**".

Now you need a message to decode. We've given you a sample coding form as APPENDIX C, with the first block already filled in and the values for the first block written in the second block. The binary numbers from Ø to 63 are entered vertically beginning at the left. When all 16 columns are filled, the numbers are read horizontally and entered into six consecutive memory cells. We suggest you use *(64) through *(69) for the example. Enter the six numbers shown into those memory locations, then type RUN, hit GO and enter -80 for X, 43 for Y and 64 for G. The 64 means that text starts in memory cell 64. (If you want to do 16 more characters, the next block would start in *(70) and go to *(75), etc., etc.)

There, see? The six numbers you entered produced 16 alphanumeric characters. You just can't pack discrete English text any tighter than that.

As for codes, try sending those six digits to the CIA ... we'll bet it keeps 'em up for a few nights!

We're sure that anyone intending to use this technique for long text-based programs will consider doing two things:

1. Write a program for generating high-density text blocks. (Just use the one at line 200 backwards.)

2. Put the Decoder into machine language for speed.

Best of luck! We expect to see some inventive programs using mixed type fonts appearing soon.

APPENDIX A

3 x 5 CHARACTER SET DATA BASE

HALF ASCII CODE	CHAR-ACTER	DATA	HALF ASCII CODE	CHAR-ACTER	DATA	HALF ASCII CODE	CHAR-ACTER	DATA
Ø	NULL	* Ø-NO ADVANCE	22	4	23497	44	J	4714
1	ERASE	Ø-DECREMENT	23	5	31118	45	K	23861
2	SPACE	Ø-INCREMENT	24	6	27119	46	L	18727
3	!	9346	25	7	29348	47	M	24429
4	"	23040	26	8	31727	48	N	27501
5	[	* 26918	27	9	31691	49	O	31599
6	↑	* 11922	28	:	1040	50	P	27556
7	%	17057	29	;	1044	51	Q	11131
8	]	* 12875	30	<	5393	52	R	27565
9	'	10240 (apos)	31	=	3640	53	S	14798
10	(	10530	32	>	17492	54	T	29842
11	)	8778	33	?	25218	55	U	23407
12	*	21845	34	↓	* 9402	56	V	23402
13	+	1488	35	A	11245	57	W	23421
14	,	20 (comma)	36	B	27566	58	X	23213
15	-	448	37	C	14627	59	Y	23186
16	.	2	38	D	27502	60	Z	29351
17	/	672	39	E	31143	61	CARRIAGE RETURN	* 0
18	Ø	11114	40	F	31140	62	x	* 2728 (mult)
19	1	11415	41	G	14699	63	÷	* 8642
20	2	25255	42	H	23533			
21	3	29647	43	I	9362			

* These eight characters must be interpreted by the decoder program. All others are equal to standard ASCII code minus 30.

```
  1 . 3 BY 5 CHARACTER GENERATOR
  2 . BY C. J.  ANDERSON / HOOVER-ANDERSON RESEARCH AND DESIGN
  3 . SET START COORD:  X, Y=UPPER LEFT PIXEL OF 1ST CHAR.  G=HI-DENSITY *( ) START
  4 INPUT X, Y, G: CLEAR
  5 . KEYBOARD INPUT ROUTINE.  DELETE FOR EMBEDDED APPLICATIONS.
  6 A=KP: GOSUB 10: GOTO 6
  7 GOTO 100: [LISTING ROUTINE]
  8 GOTO 200: [HIGH-DENSITY PACKING DEMO]
 10 . INTERPRETER FOR EIGHT NONSTANDARD CHARACTERS
 11 A=A-30: IF A=1X=X-4
 12 IF A=61A=5
 13 IF A=64A=6
 14 IF A=63A=8
 15 IF A=66A=34
 16 IF A=68A=62
 17 IF A=69A=63
 18 IF A=-17X=-80: Y=Y-6: A=0
 19 IF (A<1)+(A>63)RETURN
 20 . DATA BASE DECODER AND DISPLAY ROUTINE
 21 B=16384: F=*(A)
 22 FOR C=YTO Y-4STEP -1: FOR D=XTO X+2: E=FcB
 23 F=RM: B=Bc2: BOX D, C, 1, 1, E+2
 24 NEXT D: NEXT C: X=X+(4b(A>1))
 25 IF X=80X=-80: Y=Y-6
 26 RETURN
 30 .
 40 .
 50 .
 60 .
 70 .
 80 .
 90 .
100 . LISTING ROUTINE
110 FOR G=-24576TO -24276STEP 2
120 H=%(G)c256: A=RM: GOSUB 10: A=H: GOSUB 10: NEXT G
130 CY=0: STOP
140 .
150 .
160 .
170 .
180 .
190 . HIGH-DENSITY (6 BIT) PACKING DECODER
200 I=16384: FOR H=0TO 15
201 J=32: A=0: FOR K=GTO G+5
202 IF H=0A=A+(*(K)<0)bJ: @(K-G)=ABS(*(K))
203 IF H A=A+@(K-G)c1bJ: @(K-G)=RM
204 J=Jc2: NEXT K: GOSUB 18
205 IF H I=Ic2
206 NEXT H
```

NOTE

Lower-case c = ÷

Lower-case b = x

±	16K	8K	4K	2K	1K	512	256	128	64	32	16	8	4	2	1		±	16K	8K	4K	2K	1K	512	256	128	64	32	16	8	4	2	1

HOOVER-ANDERSON RESEARCH AND DESIGN

APPENDIX C

±	16K	8K	4K	2K	1K	512	256	128	64	32	16	8	4	2	1							
1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	32	-	3	2	7	6	4
0	0	0	1	1	1	1	1	1	0	1	0	0	0	0	0	16			8	0	9	6
0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	0	8		1	6	4	5	6
1	1	0	1	1	1	1	0	0	0	0	1	0	0	0	0	4	-	2	4	0	8	0
0	1	1	1	0	1	0	0	1	1	0	0	1	1	1	1	2		2	9	9	0	3
1	0	1	1	1	0	0	1	0	0	1	0	1	1	1	1	1	-	1	4	6	3	9