Table of Contents

Displaying a Time Gage

Displaying a Speed Gage

How fast is your program? That is a question every programmer wants to know.

A standard television updates the screen 60 times a second. The period when the beam is moving from the bottom, back up to the top is called the Vertical Blanking Period. The HuC6270 is normally set to cause an interrupt at the beginning of this period (VSYNC).

Programs for the Hu7 system generally use this interrupt as a master timer for the game.

```
main_loop:

     jsr  vsync_wait      ;Wait for VSYNC

               ;Update sprites and background, etc.

     bra  main_loop
```

The main loop should require less than 1/60 second. If it is too slow, the sprites and background may not be displayed correctly.

The HuC6280 runs at about 7.19 M Hertz (million cycles / second). In one second, there are 60 VSYNC pulses. Thus there are 119,904 machine cycles per VSYNC. This means that the main loop must be shorter than about 120,000 cycles in order to finish in 1/60 second.

You could go through your program and look up the number of cycles required for each instruction in the HuC6280 Software Manual, but there is a lot easier way to check the speed of your main loop.

We can write a very small program that will display a color bar on the side of the screen that represents the amount of time our main loop requires for each VSYNC cycle, much like a stereo displays decibels for music. As we handle more sprites and write more background data, we can see the amount of time required increase.

The method involves setting the "border color" in the HuC6260 Video Color Encoder. There are a total 32 blocks stored in the Color Table RAM. Each block has 16 colors. The first 16 blocks are for background data, the last 16 blocks are for sprite data (see H6-4). Color 0 of the first sprite block (block 16) is the border color.

Note that color 0 of block 0 (the first background block) is the background color of the screen. Color 0 of all the remaining 31

blocks are always "clear". So changing color 0 of block 16 will not effect the color of any sprites.

The border color appears around the edge of the display screen. The width of the screen is set with the HDW field of the HDR register (see H7-10). Normally we set this field to $1f, which corresponds to 32 characters across the screen. If we set the width of the screen a little narrow, we can see the border color better.

For example, we can set the screen to 31 characters wide by setting HDW to $1e. This will make the right 8 pixels of the screen blank. The color of this blank area is set with the border color (color 0 of block 16).

The HuC6260 uses 9 bits for color data. It stores the data in a word of memory in the Color Table RAM as follows:

```
xxxx xxxG GGRR RBBB
---- ---- ---- ----
```

The Color Table RAM is addressed by word. Each block requires 16 words. The first 16 blocks of background color data require 16 * 16 = 256 = $0100 words. This is the CTR address of color 0 of block 16.

The HuC6260 registers are located at physical addresses starting at $1fe400. We access them with logical addresses starting at $0400 when MPR0 = $ff (see H8-9).

If we want our color bar to be blue, for example, the program could look as follows:

```
TEKA_ADDR_LOW       equ  $0402      ;Log addr of Teka addr
TEKA_ADDR_HIGH      equ  $0403      ;
TEKA_DATA_LOW       equ  $0404      ;Log addr of Teka data
TEKA_DATA_HIGH      equ  $0405      ;

CTR_BC_ADDR         equ  $0100      ;Color Table RAM address
                                    ;of Border Color
COLOR_BLUE          equ  $0007      ;Color data.

set_bc_blue:
;
; Set HuC6260 (Tekkannon) Address to first color of block
; 16 (Border Color).
;
     lda  #low CTR_BC_ADDR
     sta  TEKA_ADDR_LOW

     lda  #high CTR_BC_ADDR
     sta  TEKA_ADDR_HIGH
```

```
;
; Set the border color to blue.
;
     lda  #low COL_BLUE
     sta  TEKA_DATA_LOW

     lda  #high COL_BLUE
     sta  TEKA_DATA_HIGH

     rts
;
; Set border color to black.
;
set_bc_black:

     lda  #low CTR_BC_ADDR
     sta  TEKA_ADDR_LOW

     lda  #high CTR_BC_ADDR
     sta  TEKA_ADDR_HIGH

     cla
     sta  TEKA_DATA_LOW
     sta  TEKA_DATA_HIGH
     rts
```

We set the border color to blue after we return from the VSYNC handle routine. The border color remains blue as our program to update sprites and background runs. When the program finishes, we come back to the beginning of the main loop. Then we set the border color back to black. The color remains black while we are waiting for the VSYNC handle routine to finish.

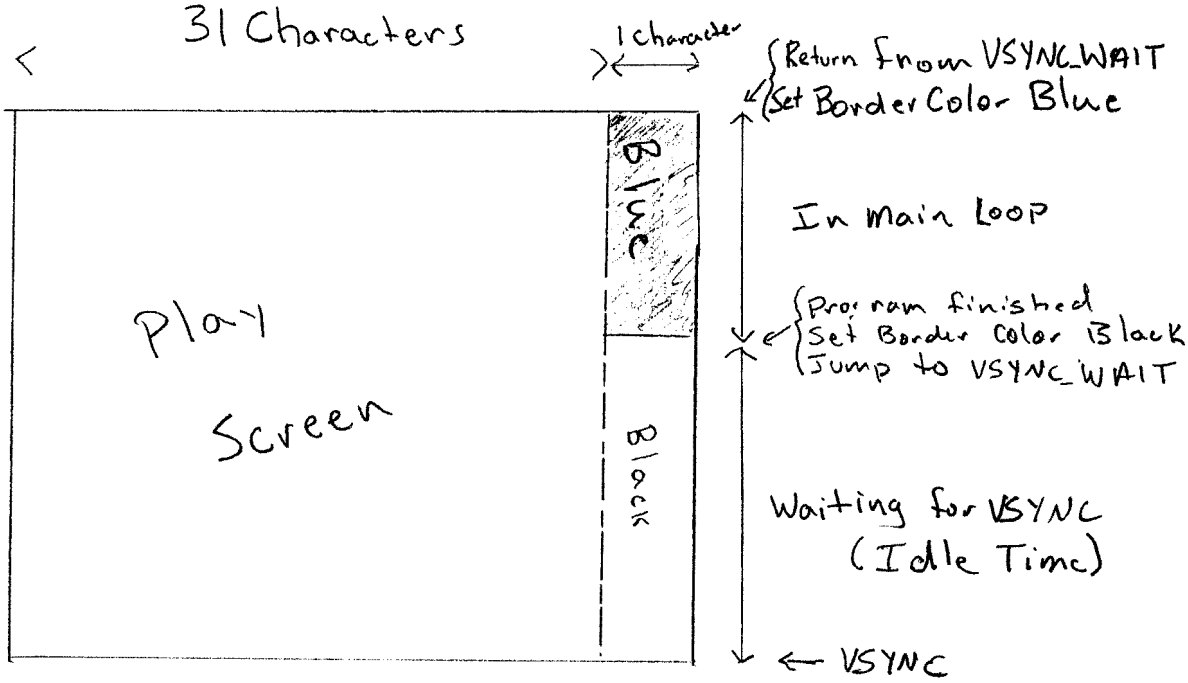
```
main_loop:

     jsr  set_bc_black        ; Set border color to black
     jsr  vsync_wait          ; Wait for VSYNC
     jsr  set_bc_blue         ; Set border color to blue

               ; Update sprites and background, etc.

     bra  main_loop
```

The screen height represents one VSYNC period (1/60 second). If the blue bar gets down to the bottom of the screen, we know that our main loop is too slow.



We can test our "speedometer" by calling a routine from our main loop to waist time.

```
main_loop:

        jsr  set_bc_black       ; Set border color to black
        jsr  vsync_wait         ; Wait for VSYNC
        jsr  set_bc_blue        ; Set border color to blue

        jsr  waist_time

        bra  main_loop
```

<waist_time> will execute a block transfer (tai) to fill an unused bank with zeros. In this example I am using bank $10.

```
_zero     dw   $0000

waist_time:
;
; Save MPR 2 and set it to bank $10 ($020000 physical)
;
          tma2
          pha

          lda  #$10
          tam2
;
; Fill bank $10 with zeros.
;
          tai  _zero,$4000,$2000
;
; Restore MPR 2 and return
;
          pla
          tma2

          rts
```

The tai instruction requires 17 + 6x cycles, where x is the number of bytes transferred (see S8-89). Here we are transferring $2000 = 8,192 bytes. So the instruction requires 49,169, or nearly 50,000 cycles. Recall that in one 1/60 second VSYNC cycle there are about 120,000 machine cycles, so this transfer takes up nearly half of the display cycle. On the screen, we can see the color bar come ~~down~~ almost half way down.

Here's another method to determine if our main loop is longer than 1/60 second. Normally, we use a flag to tell our VSYNC wait routine that the VSYNC interrupt has occurred and the VSYNC handle routine has executed.

```
vsync_wait:
     rmb0      vsync_flag

vsync_wait_loop:
     bbr0      vsync_flag,vsync_wait_loop

     rts

vsync_handle:
;
; handle the vsync
;

     smb0      vsync_flag
     rti
```

But if our main loop is too long, the VSYNC interrupt will occur somewhere in our program, before we call <vsync_wait>. When we do finally call <vsync_wait>, we may have to wait almost 1/60 before the next VSYNC.

We can modify the above method to tell us if the VSYNC interrupt occurred before we got to <vsync_wait>. Instead of a flag, we will use two counters.

```
vsync_wait:
;
; Check if the main loop was too long.
;
     lda       vsync_count
     cmp       vsync_old_count
     bne       main_loop_too_long
;
; Main loop was OK.
; Wait for vsync handle routine to finish.
;
vsync_wait_loop:
     lda       vsync_count
     cmp       vsync_old_count
     beq       vsync_wait_loop
;
; The vsync handle routine has finished.
; Set old counter to current counter.
;
     sta       vsync_old_count
     rts
;
; The main loop was too long!
;
main_loop_too_long:
     sta       vsync_count
     bra       vsync_wait_loop



vsync_handle:
;
; Handle the vsync.
;

inc       vsync_count
rti
```

In our initialization routine (reset) we should set <vsync_count> and <vsync_old_count> to zero. Normally, when we get to <vsync_wait>, <vsync_count> and <vsync_old_count> are the same. In this case, we just wait in the loop until <vsync_count> is incremented in <vsync_handle>.

But if the main loop is too slow, a VSYNC interrupt will occur in the middle of our program. The <vsync_handle> routine will run, and <vsync_count> will be incremented. When we get to <vsync_wait>, <vsync_count> will be one greater than <vsync_old_count>.

Note that in many cases, for example during program initialization, it is OK if a VSYNC interrupt occurs in our program. In this case, we can continue waiting for the next VSYNC.

From SD, run your program until you get to a stage in which you think the main loop might be too slow. Then hit a key to halt the program. Set a break point at <main_loop_too_slow> and continue running the program. If your main loop is too slow, SD will halt at the break point.

With this method of using two counters, it is possible to utilize all of the VSYNC cycle, without wasting time waiting for the VSYNC handle routine to run.

Displaying Status Lines

Displaying Status Lines

Unlike the SuperGrafx and other game machines, the TurboGrafx has only one background screen. This makes displaying a status line on a scrolling background a little difficult.

There are many methods to display status lines on a scrolling background, but they all involve setting the Scan Line Detection Register on the Video Display Controller (HuC6270) to cause a raster interrupt, and setting the Scrolling Registers to display the area of VRAM where the status line is actually written.

Note that in many scrolling games, the game status is displayed with sprite data (e.g. "Bonk") This is much easier than using BG data, and it only uses a small part of the screen, making most of the screen visible for playing.

But for many games (e.g. "J.J. and Jeff"), a status line is preferable.

In this example, we will assume that we want to make a horizontal scrolling background game (Side-Scroller) with a status line on the bottom of the screen. We will not worry about vertical scrolling for the moment. Further, we will use the 4 by 1 screen mode (SCREEN = 2 in Memory Access Width Register, see H7-9) so that we do not have to write the background data while the player is moving.

Note that in the 4 by 1 screen mode, the CG area of VRAM requires $1000 words.

With this method, when the player enters an area, the four screens are written all at once. The disadvantage of this system is that backgrounds are limited to only 4 screens. The advantages are that it is very simple to program, and the background is only written when the player enters a new area. This frees the processor while the player is moving in the area to move sprites.

When the player enters a new area:

1. Turn the screen off.
2. Write four screens of background
3. Turn the screen back on
4. Move in the 4-screen area until player exits.
5. Go to 1

We normally set the Control Register (CR) to interrupt our program when the beam begins moving from the bottom of the screen back to the top (VSYNC). But we can also set it to interrupt

when the beam is moving from the right of the screen, back to the left of the next line (HSYNC). We specify which line to interrupt on with the RCR register (see H7-7).

The system keeps an internal scanning line counter, which counts the lines on the screen as the beam sweeps from the top to bottom. When the value of RCR matches this counter, an interrupt (IRQ1) will occur, and bit 2 of the Status Register will be set (See H7-4).

In our IRQ1 handle routine, we must look at the Status Register (SR) to see whether the interrupt was caused by VSYNC or HSYNC (see H7-4).

```
AR          equ  $0000      ;log address of HuC6270 Address Reg
SR          equ  $0000      ;log address of HuC6270 Status Reg
7UP_LOW     equ  $0002      ;log address of HuC6270 low data
7UP_HIGH    equ  $0003      ;log address of HuC6270 high data

irq1_handle:
;
; Save the environment.
;
          pha
          phx
          phy
;
; Check SR to see whether VSYNC or HSYNC caused interrupt.
;    bit 2 - Scanning Line Detect (HSYNC)
;    bit 5 - Vertical Blanking Period Detect (VSYNC)
;
          lda  SR              ;load HuC6270 Status Register
          sta  sr_buf          ;save in zero page.

          bbs2 sr_buf,hsync_handle      ;check bit 2
          bbs5 sr_buf,vsync_handle      ;check bit 5
;
; Otherwise, something else caused interrupt!
;
     if   DEBUG
          brk
          nop
     endif
```

<sr_buf> is a byte of zero page memory. Zero page memory is handy because we can use the bbs instructions.

We want to display our status line on the bottom of the screen. Normally, we set the screen resolution to 240 lines horizontally (VDW field of VDR = $ef, see H7-10 and H7-13). This means there are 240 / 8 = 30 = $1e character lines on a screen. So the bottom character line is at $1d.

But if we write our status line at $1d, it will be down off the bottom of the screen. So we should move it up a couple of lines to $1b, and make the last two character lines blank.

Note that we can test this position in CE by getting into the BG mode and writing horizontal lines at $1b to $1d, then set OUT on.

We want the system to interrupt at the beginning of character line $1b. This is scan line $1b * 8 = $d8. But the Hu7 always sets the internal scanning line counter to 64 at the beginning of a vertical sweep (see H7-7). This means that if we set the RCR to a number less than 64, a raster interrupt will never occur.

Thus we should always add 64 = $40 to the scan line number that we want to interrupt on. In our case, we want to interrupt on scan line $d8. So we should set RCR to $d8 + $40 = $118.

```
set_reg   RCR
set_data  $0118
```

When we want to disable the raster interrupt, for example when we the player enters a new area and we want to write a new background, we can set RCR to zero.

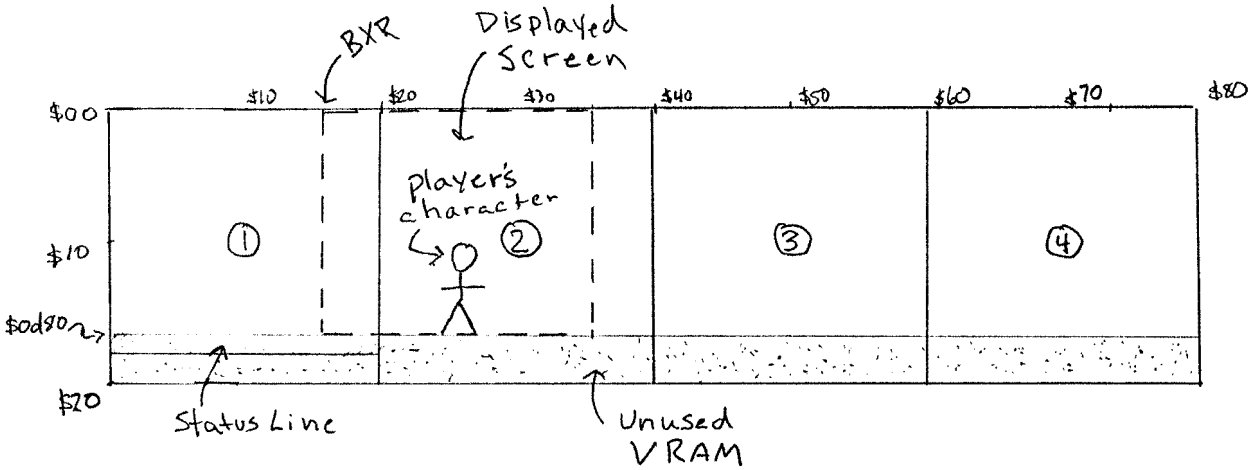
We can actually write the status line anywhere in the BAT area of VRAM that is not being used for the background. When we hit our raster interrupt, we will set the BXR and BYR scroll registers to display this area of VRAM at the bottom of the screen.

For this example, we will write our status line at character position x=$00, y= $1b. This is where it would go if we were not scrolling the background. Since we are using the 4 by 1 screen mode, there are 4*32 = 128 = $80 characters on a line. So line $1b will start at VRAM address $1b*$80 = $0d80.

Note that you can write into this area of VRAM from SD with the VF command. For example, you can load a bank of characters starting at VRAM address $1000. If you have drawn something in character number 1 with CE, you can put it on the status line, with color 0, by setting VRAM as follows (see H7-14):

```
>VF d80 120 0101
```

You can also use the VE and VD commands to edit and dump this area of VRAM.



In our interrupt handle routine, we simply set the BGX Scroll Register to zero. This will have the effect of putting the screen back at the beginning of the area.

The BGY Scroll register is a little more complicated. We want to display the area of VRAM starting at character line $1b. This is scan line $1b*8 = $d8. Thus we should set BGY to $00d8. But because we are updating BGY during display (not during VBLANK), we should set BGY to $00d8-1 (see H7-7).

We also must turn off the all the sprites when the beam gets down to the status line so that sprites are not displayed "in front" of the status line.

Thus the code in our IRQ1 handle routine could look as follows:

```
STAT_BGX        equ         $0000
STAT_BGY        equ         $00d8-1

hsync_handle:
;
; Set scroll registers to display area of VRAM where
; the status line has been written.
;
          st0       #BGX
          st1       #low        STAT_BGX
          st2       #high       STAT_BGX

          st0       #BGY
          st1       #low        STAT_BGY
          st2       #high       STAT_BGY
;
; Turn off sprites.
;
          rmb6      cr_buf            ;reset bit 6 of CR
          lda       cr_buf
          st0       #CR
          sta       7UP_LOW
          bra       irq1_exit
```

We have reserved a word of zero page memory as a Control Register buffer <cr_buf>. This is because it is impossible to read the current value of the CR in the HuC6270. It is only possible to read the Status Register.

When we exit from the IRQ1 handle routine, we should always restore the AR in case the interrupt occurred between a <set_reg> and a <set_data> instruction.

```
;
; Reset the HuC6270 Address in case interrupt occurred
; between a <set_reg> and a <set_data> instruction.
;
irq1_exit:
          lda  ar_buf
          sta  AR
;
; Restore the environment.
;
          ply
          plx
          pla

          rti
```

When the beam gets down to scan line 216 = $d8, the HSYNC interrupt handle routine will set our scroll registers to display the area of VRAM where we wrote our status line.

Now when the beam goes from the bottom of the screen back to the top to start a new vertical sweep, we need to reset the scroll registers back to their game values, and turn the sprites back on.

```
vsync_handle:
;
; Set X Scroll Register to game value.
;
          st0       #BGX
          lda       bgx_val
          sta       7UP_LOW
          lda       bgx_val+1
          sta       7UP_HIGH
;
; Set Y Scroll Register to game value.
;
          st0       #BGY
          lda       bgy_val
          sta       7UP_LOW
          lda       bgy_val+1
          sta       7UP_HIGH
```

```
;
; Turn the sprites back on
;
        smb6    cr_buf          ;set bit 6 of CR
        lda     cr_buf
        st0     #CR
        sta     7UP_LOW

        bra     irq1_exit
```

Here <bgx_val> and <bgy_val> are defined in DSEG and set in the game as the player moves around the area.

This system works OK if you do not want to do any vertical scrolling. Note, however, that we can get at least two screens of vertical scroll very easily by using the 4 by 2 screen mode (SCREEN field of MWR = 6, see H7-9). The CG area will now require $2000 words of VRAM.

We should write our status line somewhere out of the way in the CG area. For example, we could place it at the bottom three lines of page 5. There are $20 lines per page, so there are $40 lines in all. The status line should be written at lines $3d to $3f.

Note that the RCR setting will be the same, as we still want the status line to appear at the same place on the screen. Also, STAT_BGX will still be zero because we placed the status line in column $00.

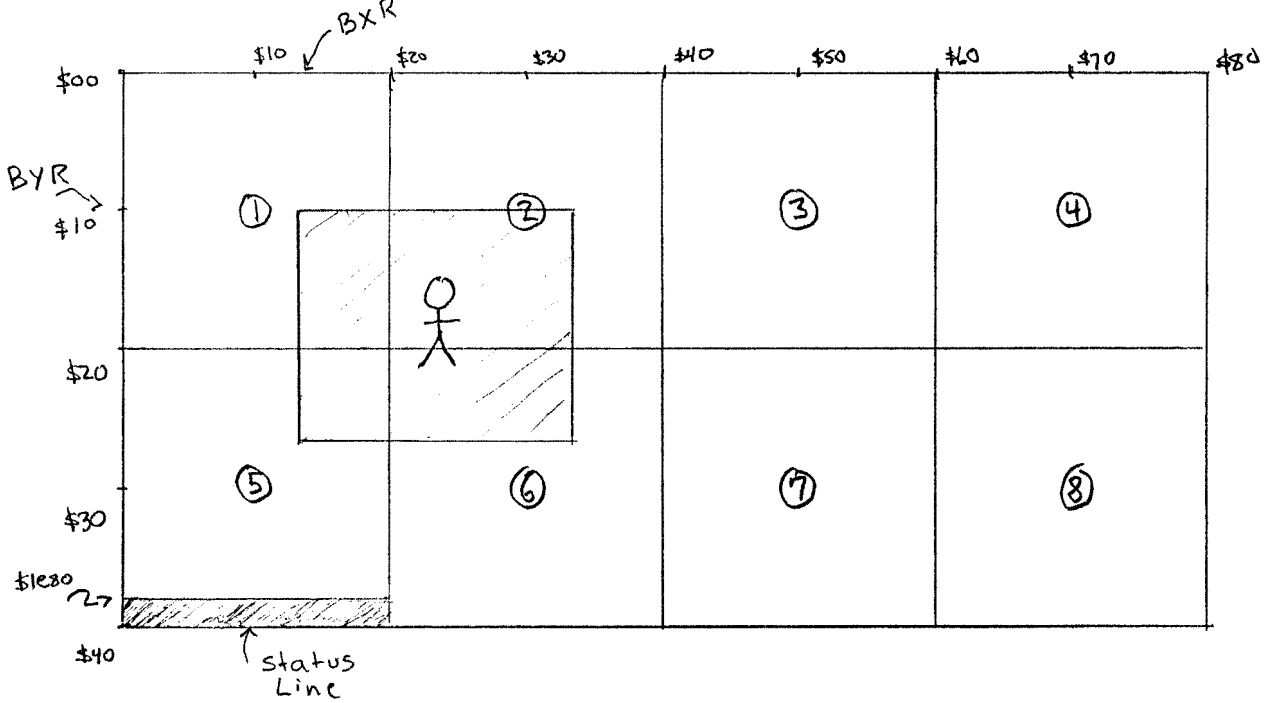
But we have moved the status line down to character line $3d. This is scan line number $3d * 8 = $01e8, so we should set STAT_BGY to $01e8-1. The VRAM address of the status line will be at $3d * $80 = $1e80.

Note that now we can use all available VRAM except for the three lines where the status line is written.

The height of the display screen is 27 characters = 27 * 8 = 216 lines. With two vertical pages, there are a total of 32 * 8 * 2 = 512 lines of VRAM to scroll in. 3 character lines , or 3 * 8 = 24 lines are used for the status line. Thus we can scroll 512 - 216 - 24 = 272 lines vertically when the display screen is over the status line area of VRAM.

But when BGX is equal to or greater than $20, we do not have to worry about the status line at the bottom, so we can scroll the full 512 - 216 = 296 lines.

This gives the player a fairly big area in which to move around. If you want to scroll more than this, you will have to update the background while the player is moving. These algorithms are a little more tricky, especially if you want to display a status line.


BXR
$10
$20
$30
$40
$50
$60
$70
$80
$00
BYR
$10
$20
$30
$1e30
$40
1
2
3
4
5
6
7
8
status
Line

Displaying Japanese Characters

Displaying Japanese Characters

Game developers who want to sell their games in Japan may want to use Japanese characters for their messages. Though it is possible to write the Japanese language with the Roman alphabet ("Romaji"), it is difficult for Japanese people to read. Thus Japanese characters are generally preferred.

There are three kinds of Japanese characters. "Hiragana" is a kind of alphabet. "Katakana" is equivalent to Hiragana, but is used for writing foreign (e.g. English) words. "Kanji" are Chinese characters.

For most games that do not have a lot of messages, Hiragana and Katakana only can be used. These characters can be made 8 dots by 8 dots, and can fit into a half bank (128 characters).

We then make an include file (JAP.H) that defines the character numbers:

```
a           equ  0
i           equ  1
u           equ  2
e           equ  3
o           equ  4
ka          equ  5
ki          equ  6
ku          equ  7
ke          equ  8
ko          equ  9
:
```

On down to 128. Then we make a message file (MESG.S):

```
mesg00:
     db   ko,n,ni,chi,wa
```

This will say "konnichiwa" (hello) in Japanese. At the bottom of the file, we make a table of the message labels:

```
mesg_tbl:
     dw   mesg00
     dw   mesg01
     :
```

For however many messages we have. Then we must make a simple display routine that, given the message number, will display the message in Japanese on the screen.

こんにちわ

Many PC Engine (HuCard) games use some Kanjis. Character sizes are usually 16-by-16 or 12-by-12 dots. 64 16-by-16 characters can fit into one bank. 100 12-by-12 characters can fit into one bank. One bank requires 8 K bytes.

Some games use the remaining half-bank of characters used for Hiragana and Katakana. You can fit as many as 50 kanjis in half a bank. Some HuCard games use as many as 256 Kanjis. This requires 4 banks for 16-by-16 characters.

Of course character data to store Kanjis can be compressed like any other kind of data. It is often possible to compress character data 50 percent.

For CD games that have a lot of messages (e.g. RPG games), it is better to use more Kanji. There are many thousands of Kanjis used in the Japanese language.

The TurboGrafx CD-ROM System stores the entire 8000 or so characters that make up the JIS (Japanese Industrial Standards) Level 1. CD allows games to use so much memory.

The JIS system is convenient because it is used in Japanese word processors. Thus defining each character like we did for Hiragana and Katakana is not necessary. We can simply use a Japanese word processor to make the message file. Japanese word processors output JIS "Shift" Code. The attached table shows the first of eight pages of JIS codes.

The <EX_GETFNT> routine in CD-ROM BIOS reads kanji character data from the CD. The input is the JIS Shift Code. For example, JIS Shift Code $938c will return a font pattern that looks something like the following:

```
     0123456789abcdef
0    0000000100000000
1    1111111111111110
2    0000000100000000
3    1111111111111110
4    1000000100000010          東
5    1000000100000010
6    1111111111111110
7    1000000100000010
8    1000000100000010
9    1111111111111110
a    0000000100000000
b    0000001110000000
c    0000010101000000
d    0000100100100000
e    0010000100001000
f    1000000100000010
```

This kanji means "East", symbolized by the red sun rising behind a tree.

At Hudson, we have services available for all your translation, localization, and voice recording needs. We will be happy to assist you in your translation projects as our staff is fully competent in Japanese, English, French, Spanish, and German.

JIS "Shift" Code →

Kanji Code Table (JIS Level 1)

漢字コード表（JIS第1水準）

	シフトJIS	JIS	0 1 2 3	4 5 6 7	8 9 A B	C D E F
Miscellaneous 記号	813F	2120	[SP]、。	，．・：	；？！゛	゜´ ` ¨
	814F	2130	^ ￣ _ ヽ	ヾ ゝ ゞ 〃	仝 々 〆 〇	― ‐ ／ ＼
	815F	2140	～ ∥ ｜	… ‥ ‘ ’	“ ” （ ）	〔 〕 ［ ］
	816F	2150	｛ ｝ 〈 〉	《 》 「 」	『 』 【 】	＋ － ± ×
	8180	2160	÷ ＝ ≠ ＜	＞ ≦ ≧ ∞	∴ ♂ ♀ °	′ ″ ℃ ￥
	8190	2170	＄ ¢ £ ％	＃ ＆ ＊ ＠	§ ☆ ★ ○	● ◎ ◇
	819E	2220	◆ □ ■	△ ▲ ▽ ▼	※ 〒 → ←	↑ ↓ 〓
Numbers, Roman Alphabet 英・数字	824F	2330	０ １ ２ ３	４ ５ ６ ７	８ ９	
	825F	2340	Ａ Ｂ Ｃ	Ｄ Ｅ Ｆ Ｇ	Ｈ Ｉ Ｊ Ｋ	Ｌ Ｍ Ｎ Ｏ
	826F	2350	Ｐ Ｑ Ｒ Ｓ	Ｔ Ｕ Ｖ Ｗ	Ｘ Ｙ Ｚ	
	8280	2360	ａ ｂ ｃ	ｄ ｅ ｆ ｇ	ｈ ｉ ｊ ｋ	ｌ ｍ ｎ ｏ
	8290	2370	ｐ ｑ ｒ ｓ	ｔ ｕ ｖ ｗ	ｘ ｙ ｚ	
Hiragana ひらがな	829E	2420	ぁ あ ぃ	い ぅ う ぇ	え ぉ お か	が き ぎ く
	82AE	2430	ぐ け げ こ	ご さ ざ し	じ す ず せ	ぜ そ ぞ た
	82BE	2440	だ ち ぢ っ	つ づ て で	と ど な に	ぬ ね の は
	82CE	2450	ば ぱ ひ び	ぴ ふ ぶ ぷ	へ べ ぺ ほ	ぼ ぽ ま み
	82DE	2460	む め も ゃ	や ゅ ゆ ょ	よ ら り る	れ ろ ゎ わ
	82EE	2470	ゐ ゑ を ん			
Katakana カタカナ	833F	2520	ァ ア ィ	イ ゥ ウ ェ	エ ォ オ カ	ガ キ ギ ク
	834F	2530	グ ケ ゲ コ	ゴ サ ザ シ	ジ ス ズ セ	ゼ ソ ゾ タ
	835F	2540	ダ チ ヂ ッ	ツ ヅ テ デ	ト ド ナ ニ	ヌ ネ ノ ハ
	836F	2550	バ パ ヒ ビ	ピ フ ブ プ	ヘ ベ ペ ホ	ボ ポ マ ミ
	8380	2560	ム メ モ ャ	ヤ ュ ユ ョ	ヨ ラ リ ル	レ ロ ヮ ワ
	8390	2570	ヰ エ ヲ ン	ヴ ヵ ヶ		
Greek ギリシア文字	839E	2620	Α Β Γ	Δ Ε Ζ Η	Θ Ι Κ Λ	Μ Ν Ξ Ο
	83AE	2630	Π Ρ Σ Τ	Υ Φ Χ Ψ	Ω	
	83BE	2640	α β γ	δ ε ζ η	θ ι κ λ	μ ν ξ ο
	83CE	2650	π ρ σ τ	υ φ χ ψ	ω	
Russian ロシア文字	843F	2720	А Б В	Г Д Е Ё	Ж З И Й	К Л М Н
	844F	2730	О П Р С	Т У Ф Х	Ц Ч Ш Щ	Ъ Ы Ь Э
	845F	2740	Ю Я			
	846F	2750	а б в	г д е ё	ж з и й	к л м н
	8480	2760	о п р с	т у ф х	ц ч ш щ	ъ ы ь э
	8490	2770	ю я			
Kanji "A" ア	889E	3020	亜 唖 娃	阿 哀 愛 挨	姶 逢 葵 茜	穐 悪 握 渥
	88AE	3030	旭 葦 芦 鯵	梓 圧 斡 扱	宛 姐 虻 飴	絢 綾 鮎 或
	88BE	3040	粟 袷 安 庵	按 暗 案 闇	鞍 杏	
	シフトJIS	JIS	0 1 2 3	4 5 6 7	8 9 A B	C D E F

Using the SET_REG Macro

Using the SET REG macro

Most Hu7 programs define a macro to set the register on the Video Display Controller (HuC6270).

```
set_reg   MACRO     reg_num
          lda       #reg_num
          sta       ar_buf
          st0       #reg_num
          ENDM
```

In our program, if we want to write data to VRAM address $1000, for example, we would select the Memory Address Write register and set it to $1000.

```
MAWR      equ       0          ;HuC6270 write register

          set_reg   MAWR       ;set register
          st1       #$00       ;send low byte
          st2       #$01       ;send high byte
```

But why do we have to save the register number in <ar_buf>? Lets say we are not using the <set_reg> macro. In our program, we would select the MAWR register to $1000 as follows:

```
          st0       #MAWR
          st1       #$00
          st2       #$01
```

It seems like this code would do the same thing. But the problem comes up when a HuC6270 interrupt (HSYNC, VSYNC, etc, see IE field of CR, page H7-6) occurs after the <st0> and before the <st1> or <st2> commands.

```
          st0       #MAWR
                              <INTERRUPT>
          st1       #$00
          st2       #$01
```

If our interrupt handle routine sets the HuC6270 register, which it probably will, the <st1> and <st2> commands will send data to the wrong register.

Therefore, when we exit from our interrupt handle routine, we should always reset the HuC6270 Address Register.

```
AR        equ       $0000      ;log addr of HuC6270
                               ;address register

irq1_exit:
;
; Reset HuC6270 Address Register.
;
          lda       ar_buf
          sta       AR
```

```
;
; Restore the environment.
;
          ply
          plx
          pla

          rti
```

Note that here, we are setting the HuC6270 address register by sending the contents of the A register to logical address $0000. This is physical address $1fe000 when MPR0 = $ff.

This demonstrates the two methods for setting the HuC6270 registers. Usually we use <st0> to set the register number and <st1> and <st2> to set the low and high byte of the register. But these commands only work with immediate data (see S8-80 to 82).

Thus when we must set the HuC6270 register from a memory value, we must use logical location $0000 for the register number and $0002 and $0003 for the high and low byte.

You must be careful not to use the <set_reg> macro inside the interrupt handle routine itself, as this will reset the <ar_buf>. This may cause very strange bugs in your program. Inside the interrupt handle routine, you should set the register number directly either with <st0> or using location $0000.

Packing Color Data

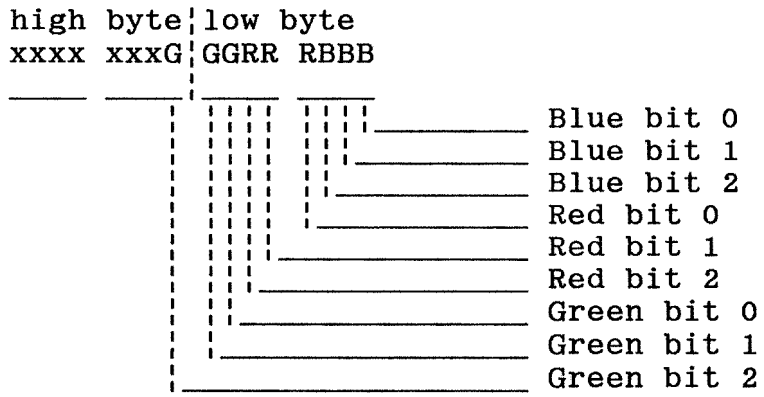
Packing Color Data

Color data for both sprite and background data from the CE paint program is stored in .CCD (or .CCB or CCH) files. There are 16 pallets, each with 16 colors, in a .CCD file.

For example, the data for one color pallet might look like the following:

```
        dw      $0000,$0038,$00F8,$01F8,$01D8,$01C0,$01C3,$01C7
        dw      $00C7,$0007,$001F,$003F,$003B,$0092,$0124,$01FF
```

The red, green and blue components are assigned 3 bits each. The 9 bits are stored in a word of memory with the following format:



Note that only bit 0 of the high-order byte is used. The seven other bits are always zero. We can pack this data by storing Green bit 2 for eight of the colors in a separate byte.

```
                db      $1f,$00,$38,$f8,$f8,$d8,$c0,$c3,$c7
                db      $03,$c7,$07,$1f,$3f,$3b,$92,$24,$ff
```

Here, each line has nine bytes. The first byte on line stores the Green bit 2. The following 8 bytes on the line are the same as the low-order bytes of the unpacked color data.

It is then very simply to unpack the data and send it to the Color Table RAM on the Video Color Encoder (HuC6260). For each group of eight colors, shift the high bit code byte. This bit is Green bit 2.

How much memory have we saved? The unpacked data requires 2 bytes for each color. One pallet of 16 colors requires 16 * 2 = 32 bytes. 16 pallets in a .CCD file require 16 * 32 = 512 bytes.

Our packed data uses 2 * 9 = 18 bytes per pallet, thus 18 * 16 = 188 bytes per .CCD file. So we are saving 512 - 188 = 324 bytes per .CCD file. If we are using one .CCD file for spite colors and one .CCD file for background colors, we save 324 * 2 = 648 bytes. The packed data will require about 63 percent as much memory as the unpacked data.