Displaying Apple II Double Hi-Res Pixel Graphics in a cc65 C Program

Table of Contents

Displaying Apple II Double Hi-Res Pixel Graphics in a cc65 C Program....1
Table of Contents....1
Introduction....2
Accessing Auxiliary Memory....3
Linker Configuration....4
Tables Used by This Demo....5
Text Screen Base Address Table....5
Graphics Screen Base Address Table....6
The Byte Table....7
The Pixel Tables....9
Plotting Pixels with Tables....10
Line Routines....13
Graphics Primitives....17
The Circle....17
The Box....19
Block Fill Routines....19
Font Routines....23
Setting Video Modes....26
Mixed Text and Graphics....27
Building the Demo....28

The WORLD Program.....29
The Main Program.....31
Additional Notes.....33
References.....33

Introduction

This document is about a Demo Program (called "World") written in C (and compiled using cc65) that plots Double Hi-Res (DHGR) Pixel Graphics on the Apple //e.

Double Hi-Res (DHGR) Pixel Graphics on the Apple //e in a C program is somewhat less complicated using DHGR Page One at $2000 than DHGR Page Two at $4000, considerably quicker, and provides more memory.

Writing a ProDOS SYS program to use DHGR Page One is impossible because the program load address of $2000 conflicts with the screen address. A cc65 SYS program that uses DHGR Page Two at $4000 has only 8192 bytes of memory, so it is impractical.

The cc65 programmer has two practical alternatives:

- BIN Program that loads at $0803 and uses DHGR Page Two at $4000 providing 14333 bytes below screen memory.

- BIN Program that loads at $4000 and uses DHGR Page One at $2000 providing 24272 bytes above screen memory.

In either case, Oliver Schmidt's tiny loader.system can be used to launch the program.

It is obvious that the BIN program that uses DHGR Page One is the logical choice on the basis of available memory alone. But before we take a look at our DHGR demo program which uses DHGR Page One, let's look at the efficiency difference that exists between these two practical choices.

Accessing Auxiliary Memory

When we use DHGR Page One, we don't usually need to buffer auxiliary screen memory so we can address the screen directly by using the following soft switches:

```
#define  dhraux  ((unsigned char*)0xC055)
#define  dhrmain ((unsigned char*)0xC054)
```

Switching between auxiliary and main DHGR screen memory in a cc65 program is as easy as writing a byte to the above soft switch addresses:

```
dhraux[0] = 0;  /* select auxiliary memory */
dhrmain[0] = 0; /* reset to main memory */
```

Or alternately by storing whatever is in the accumulator to the soft switch:

```
asm("sta $c055"); /* AUX MEM */
asm("sta $c054"); /* MAIN MEM */
```

But when we use DHGR Page Two, the AUXMOVE routine is used to access auxiliary screen memory, and buffers and additional instructions are required making the code larger and less efficient. Considering the much smaller available memory in a cc65 program that uses Page Two DHGR, the Page One alternative looks even better.

AUXMOVE is generally a handy routine for any Apple II program that stores and retrieves data in auxiliary memory:

```
/* move a block of data from main to auxiliary memory */
void maintoaux(unsigned src0, unsigned src1, unsigned dest0)
{
      unsigned *src  = (unsigned *)0x3c;
      unsigned *dest = (unsigned *)0x42;
      src[0] = src0;
      src[1] = src1;
      dest[0] = dest0;
      asm("sec");
      asm("jsr $c311");
}
```

```
/* move a block of data from auxiliary to main memory */
void auxtomain(unsigned src0, unsigned src1, unsigned dest0)
{
      unsigned *src  = (unsigned *)0x3c;
      unsigned *dest = (unsigned *)0x42;
      src[0] = src0;
      src[1] = src1;
      dest[0] = dest0;
      asm("clc");
      asm("jsr $c311");
}
```

And for some DHGR functions aside from pixel graphics it compares not too differently regardless of whether we choose to use DHGR screens at $2000 or $4000:

```
void dhiresclear(void)
{
    /* clear auxiliary screen memory */
    asm("sta $c055"); /* AUX MEM */
    memset((char *)0x2000,0,8192);
    /* clear main screen memory */
    asm("sta $c054"); /* MAIN MEM */
    memset((char *)0x2000,0,8192);
}

void dhiresclear(void)
{
    /* clear main screen memory */
    memset((char *)0x4000,0,8192);
    /* clear auxiliary screen memory */
    maintoaux(0x4000,0x4000+8191,0x4000);
}
```

In fact for loading a DHGR bitmapped graphics file to either screen address like in a slide-show program, AUXMOVE can be used exactly the same way. But for DHGR pixel graphics, AUXMOVE and buffering is not the best way to go, so the soft-switch method and the Page One DHGR screen at $2000 wins-out for this demo. It is simply a better over-all approach.

Linker Configuration

This demo uses a custom linker configuration file based on the apple2enh-system.cfg that comes with cc65. It is not terribly customized, because only a single MEMORY setting has been changed for RAM; instead of starting at $2000, the start address is set to $4000 which is directly above DHGR Screen One at $2000:

```
MEMORY {
  ZP:   define = yes, start = $0080,size = $001A;
  RAM:  file = %O,start = $4000,size = $7F00 - __STACKSIZE__;
  MOVE: file = %O,define = yes,start = $0000,size = $FFFF;
  LC:   define = yes,start = __LCADDR__,size = __LCSIZE__;
}
```

This leaves the memory below $2000 available, although it is not reserved. If we compare this to an Aztec C65 SYS program that does DHGR pixel graphics, cc65 not only has just as much memory, and inherently more efficient and smaller code, but is smaller yet because in Aztec C65 the Screen at $4000 must be used together with AUXMOVE resulting in even more efficiency loss and even larger code.

The Aztec C65 linker differs from cc65's linker in that a reserved area of memory over the screen address is compiled right into the Aztec C65 program resulting in an advantage of being able to jump over the reserved area. Since the cc65 linker can't do this, you might think that an Aztec C65 DHGR BIN program that uses soft-switches and the DHGR Screen at $2000 would be at least close to as efficient, but Aztec C65 quickly "chews-up" this advantage by its bulkier less-efficient and larger code and occupies even more available memory, without leaving the space below the program that is still available to the cc65 BIN program that loads at $4000.

Although not part of the cc65 program, the main memory below the DHGR Screen at $2000, starting at $C00 provides an "unofficial" 5120 bytes that can be used for things like storing data tables and fonts that can be loaded from disk. This demo does not do that however, and stores its tables and font in the program itself, and runs well.

Tables Used by This Demo

I am a big fan of Look Up Tables (LUTs) for pre-calculated values that simplify bit-mapped graphics and pixel graphics programming. They result in code that is a little more readable, and the memory they take-up is also generally balanced-out by code that may be faster or smaller. At least that is my hope.

As previously mentioned they can be stored in a binary format and loaded from disk into memory somewhere, including auxiliary memory. Although that isn't done in this demo, it's certainly an option in a meaningful production program.

Text Screen Base Address Table

Part of this demo used the Apple II's Mixed Text and Graphics feature to print directly to the bottom 4 lines of the text screen, directly to text screen memory in 80 column mode. The routines that do so are supported by an 8 byte table with the starting address of each of these 4 lines:

```
/* base addresses for last 4 lines of text screen memory page 1
*/
unsigned dlotextbase[4]={
    0x0650,
    0x06D0,
    0x0750,
    0x07D0};
```

Whether we are using DHGR or DLGR (double lo-res) the same routines and this same table can be used exactly the same way in Mixed Text and Graphics mode. These base addresses can actually be used as base addresses of direct text to screen in any of the Apple II Mixed Text and Graphics modes, whether they are single or double resolution modes.

Graphics Screen Base Address Table

```
unsigned char *codeline = (unsigned char *) gethibase(y);
unsigned char *tableline = (unsigned char *) HB[y];
```

Whether we are in HGR (Hi-Res) or DHGR, the same base address table array can be used for the base address of each of the Screen $2000 scan-lines. Adding 8192 to these addresses will provide the base addresses for each of the Screen $4000 scan-lines.

Using a scan-line index from 0-192 is much less code intensive than calculating these every time they are needed (the two methods are shown above). Consider the code that this table replaces (shown below in the gethibase() function) and consider the overhead of calling the code below every time we need the starting address of a scanline:

```
unsigned gethibase(int currentline)
{
  int ybase=0x2000,z,a;
  if(currentline >63) {
    if (currentline < 128) {
        ybase+=0x28;
        currentline-=64;
    }
    else {
    ybase+=0x50;
    currentline-=128;
    }
  }
  z=(currentline>>3);
  a = (z<<7)|ybase;
  return  (unsigned)((currentline - (z<<3))<<10 | a);
}

/* HGR Base Array */
/* provides base address for 192 page1 hires scanlines  */
```

```
unsigned HB[]={
0x2000, 0x2400, 0x2800, 0x2C00, 0x3000, 0x3400, 0x3800, 0x3C00,
0x2080, 0x2480, 0x2880, 0x2C80, 0x3080, 0x3480, 0x3880, 0x3C80,
0x2100, 0x2500, 0x2900, 0x2D00, 0x3100, 0x3500, 0x3900, 0x3D00,
0x2180, 0x2580, 0x2980, 0x2D80, 0x3180, 0x3580, 0x3980, 0x3D80,
0x2200, 0x2600, 0x2A00, 0x2E00, 0x3200, 0x3600, 0x3A00, 0x3E00,
0x2280, 0x2680, 0x2A80, 0x2E80, 0x3280, 0x3680, 0x3A80, 0x3E80,
0x2300, 0x2700, 0x2B00, 0x2F00, 0x3300, 0x3700, 0x3B00, 0x3F00,
0x2380, 0x2780, 0x2B80, 0x2F80, 0x3380, 0x3780, 0x3B80, 0x3F80,
0x2028, 0x2428, 0x2828, 0x2C28, 0x3028, 0x3428, 0x3828, 0x3C28,
0x20A8, 0x24A8, 0x28A8, 0x2CA8, 0x30A8, 0x34A8, 0x38A8, 0x3CA8,
0x2128, 0x2528, 0x2928, 0x2D28, 0x3128, 0x3528, 0x3928, 0x3D28,
0x21A8, 0x25A8, 0x29A8, 0x2DA8, 0x31A8, 0x35A8, 0x39A8, 0x3DA8,
0x2228, 0x2628, 0x2A28, 0x2E28, 0x3228, 0x3628, 0x3A28, 0x3E28,
0x22A8, 0x26A8, 0x2AA8, 0x2EA8, 0x32A8, 0x36A8, 0x3AA8, 0x3EA8,
0x2328, 0x2728, 0x2B28, 0x2F28, 0x3328, 0x3728, 0x3B28, 0x3F28,
0x23A8, 0x27A8, 0x2BA8, 0x2FA8, 0x33A8, 0x37A8, 0x3BA8, 0x3FA8,
0x2050, 0x2450, 0x2850, 0x2C50, 0x3050, 0x3450, 0x3850, 0x3C50,
0x20D0, 0x24D0, 0x28D0, 0x2CD0, 0x30D0, 0x34D0, 0x38D0, 0x3CD0,
0x2150, 0x2550, 0x2950, 0x2D50, 0x3150, 0x3550, 0x3950, 0x3D50,
0x21D0, 0x25D0, 0x29D0, 0x2DD0, 0x31D0, 0x35D0, 0x39D0, 0x3DD0,
0x2250, 0x2650, 0x2A50, 0x2E50, 0x3250, 0x3650, 0x3A50, 0x3E50,
0x22D0, 0x26D0, 0x2AD0, 0x2ED0, 0x32D0, 0x36D0, 0x3AD0, 0x3ED0,
0x2350, 0x2750, 0x2B50, 0x2F50, 0x3350, 0x3750, 0x3B50, 0x3F50,
0x23D0, 0x27D0, 0x2BD0, 0x2FD0, 0x33D0, 0x37D0, 0x3BD0, 0x3FD0};
```

The Byte Table

3 tables are used to operate on scan-lines directly in the DHGR Screen memory area. One of these is the byte table. When a fill is needed in one of the 16 available DHGR colors, we need to use a 4 byte table to look-up the appropriate bit mask at the screen address of the scan-line referenced by the HB[] table array listed previously:

```
unsigned char dhrbytes[16][4] = {
      {0x00,0x00,0x00,0x00},
      {0x08,0x11,0x22,0x44},
      {0x11,0x22,0x44,0x08},
      {0x19,0x33,0x66,0x4C},
      {0x22,0x44,0x08,0x11},
      {0x2A,0x55,0x2A,0x55},
      {0x33,0x66,0x4C,0x19},
      {0x3B,0x77,0x6E,0x5D},
      {0x44,0x08,0x11,0x22},
      {0x4C,0x19,0x33,0x66},
      {0x55,0x2A,0x55,0x2A},
      {0x5D,0x3B,0x77,0x6E},
      {0x66,0x4C,0x19,0x33},
      {0x6E,0x5D,0x3B,0x77},
      {0x77,0x6E,0x5D,0x3B},
      {0x7F,0x7F,0x7F,0x7F}
```

You will note that this 2 dimensional array has 16 table entries of 4 bytes. This is a 7 pixel group that straddles alternating bytes in auxiliary and main screen memory for each of the 16 colors. The 16 bitmasks in the table are in the LORES (Low Resolution) color order which is also used by DLGR. Using this order allows us to standardize the color constants throughout all of our 16 color graphics routines for the Apple II:

```
/* lo-res colors and color order */
#define LOBLACK       0
#define LORED         1
#define LODKBLUE      2
#define LOPURPLE      3
#define LODKGREEN     4
#define LOGRAY        5
#define LOMEDBLUE     6
#define LOLTBLUE      7
#define LOBROWN       8
#define LOORANGE      9
#define LOGREY        10
#define LOPINK        11
#define LOLTGREEN     12
#define LOYELLOW      13
#define LOAQUA        14
#define LOWHITE       15
```

You may have seen the re-ordered table below in the DHGR "official" color order in Apple Computer's documentation, but in this demo, and other programs that use these custom graphics routines for cc65, to be consistent in our approach, we don't bother to follow Apple Computer's confusing different order for the same colors shared by LORES and DHGR:

```
The following array is reordered to match the lores color order.
The constants defined for lores color are used as indices for
plotting of hi-res colors.
Color          aux1  main1 aux2  main2 Repeated Binary Pattern
Black           00    00    00    00    0000
Magenta         08    11    22    44    0001
Dark Blue       11    22    44    08    1000
Violet          19    33    66    4C    1001
Dark Green      22    44    08    11    0100
Grey1           2A    55    2A    55    0101
Medium Blue     33    66    4C    19    1100
Light Blue      3B    77    6E    5D    1101
Brown           44    08    11    22    0010
Orange          4C    19    33    66    0011
Grey2           55    2A    55    2A    1010
Pink            5D    3B    77    6E    1011
Green           66    4C    19    33    0110
Yellow          6E    5D    3B    77    0111
Aqua            77    6E    5D    3B    1110
White           7F    7F    7F    7F    1111
```

```
Each 4 bytes contains 7 - 4 bit pixels straddling the lower 7
bits of the aux, main, aux, main screen memory block.
```

The DHGR screen memory has another peculiarity aside from the repeated mostly different 4 bytes in the pattern. Each byte only uses 7 bits for DHGR color mode. This makes plotting these as pixels even more of chore because in order to isolate a single pixel a fairly complicated and multiple step bit masking routine is needed.

This bit masking of pixels is quite a bit harder to understand than simply filling 4 byte patterns, so to make the code a little easier to read, and also to hopefully reduce the code to something approaching efficient, 2 additional tables are used in conjunction with some reasonably well commented routines for pixel plotting with the additional hope that you might be able to make more sense out of these routines than the mostly unreadable documentation that Apple Computer provided for DHGR mode.

But don't take my word for it. You can read Apple Computer's documentation at the following link and decide for yourself. The document does contain important and useful information related to this demo and DHGR, so like it or not it is required reading:
http://apple2.boldt.ca/?page=til/tn.aiie.003

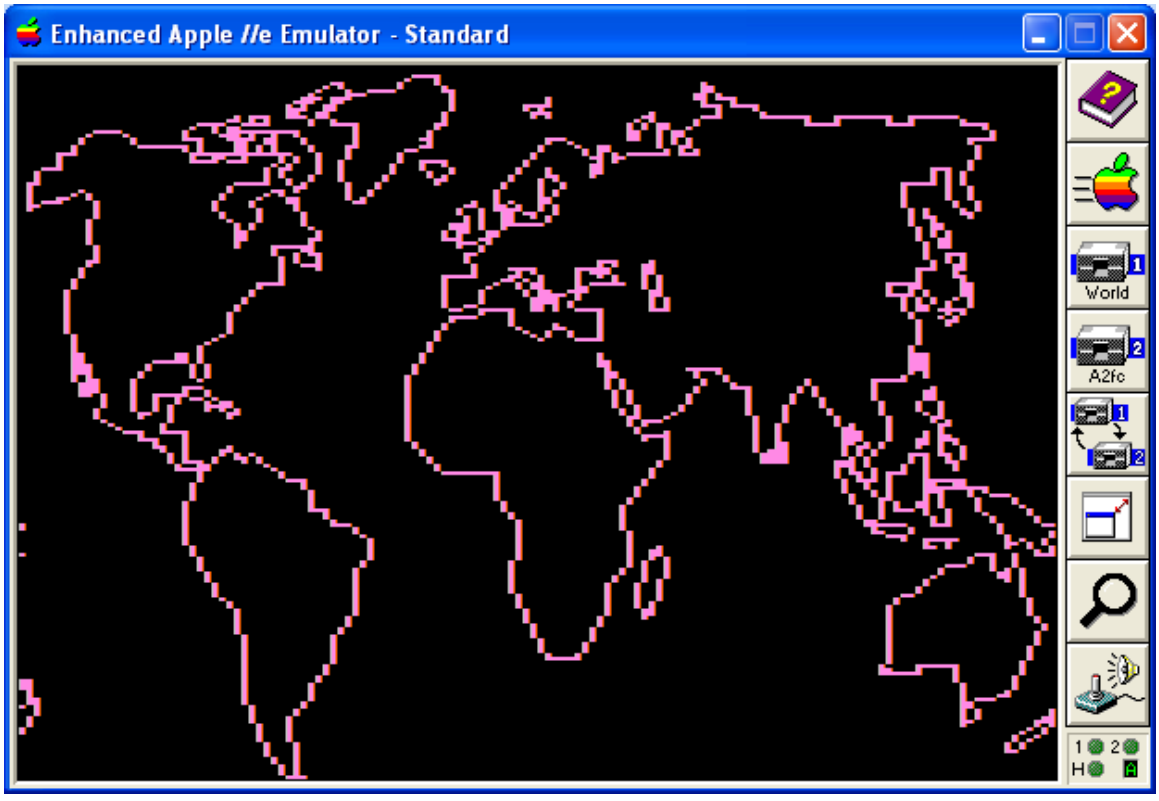
The Pixel Tables

The two tables below isolate the bitmasks for the 16 color values of each pixel for auxiliary and main memory respectively in the region of a 4 byte screen memory block. Like the other 16 color table, the color order is LORES:

```
/* The following 2 arrays contain the same 7 - 4 bit pixel values
in the array above but are split by pixels instead of bytes. Some
values are never used but provide padding for ease of table
lookup by index during the plotting of pixels. */
/* auxiliary memory */
unsigned char dhapix[16][7] = {
{0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x08,0x00,0x00,0x02,0x20,0x00,0x00},
{0x01,0x10,0x00,0x00,0x04,0x40,0x00},
{0x09,0x10,0x00,0x02,0x24,0x40,0x00},
{0x02,0x20,0x00,0x00,0x08,0x00,0x00},
{0x0A,0x20,0x00,0x02,0x28,0x00,0x00},
{0x03,0x30,0x00,0x00,0x0C,0x40,0x00},
{0x0B,0x30,0x00,0x02,0x2C,0x40,0x00},
{0x04,0x40,0x00,0x01,0x10,0x00,0x00},
{0x0C,0x40,0x00,0x03,0x30,0x00,0x00},
{0x05,0x50,0x00,0x01,0x14,0x40,0x00},
{0x0D,0x50,0x00,0x03,0x34,0x40,0x00},
{0x06,0x60,0x00,0x01,0x18,0x00,0x00},
{0x0E,0x60,0x00,0x03,0x38,0x00,0x00},
{0x07,0x70,0x00,0x01,0x1C,0x40,0x00},
{0x0F,0x70,0x00,0x03,0x3C,0x40,0x00}};
```

```
/* main memory */
unsigned char dhmpix[16][7] = {
{0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x01,0x10,0x00,0x00,0x04,0x40},
{0x00,0x00,0x02,0x20,0x00,0x00,0x08},
{0x00,0x01,0x12,0x20,0x00,0x04,0x48},
{0x00,0x00,0x04,0x40,0x00,0x01,0x10},
{0x00,0x01,0x14,0x40,0x00,0x05,0x50},
{0x00,0x00,0x06,0x60,0x00,0x01,0x18},
{0x00,0x01,0x16,0x60,0x00,0x05,0x58},
{0x00,0x00,0x08,0x00,0x00,0x02,0x20},
{0x00,0x01,0x18,0x00,0x00,0x06,0x60},
{0x00,0x00,0x0A,0x20,0x00,0x02,0x28},
{0x00,0x01,0x1A,0x20,0x00,0x06,0x68},
{0x00,0x00,0x0C,0x40,0x00,0x03,0x30},
{0x00,0x01,0x1C,0x40,0x00,0x07,0x70},
{0x00,0x00,0x0E,0x60,0x00,0x03,0x38},
{0x00,0x01,0x1E,0x60,0x00,0x07,0x78}};
```

Plotting Pixels with Tables



Bit-masking can be a confusing business in practice only because the code to do so is barely man-readable and even less man-writable... but theoretically straight-forward and quite efficient:

```
Summary of    AND and OR bitwise operators
bit a   bit b   a & b   a | b
0       0       0       0
0       1       0       1
1       0       0       1
1       1       1       1
```

- Bitwise ANDing a binary 1 will preserve the bit and Bitwise ANDing a binary 0 will erase the bit.

- Combining 2 values by inclusively ORing them together works fantastically if the areas to be combined have been mutually zeroed in the complementary bits of the other value by using a bitwise AND.

To make things even a little more interesting, in the demo program, two slightly different methods are explored. The pixel plotting function uses all 3 tables listed above, but the vertical line plotting function uses only the 4 byte table and from this, isolates the color mask on the fly. Tables are not used exclusively in either function; actual mask values are used "in the open" to erase the previous color.

Both plotting functions are quite abstract, but the single pixel plotting function is also quite well commented so you should after some study make sense of what is happening here.

First we will examine the single pixel plotting function. In this function we assign the color bitmap for the inclusive OR from the 7 pixel level color bitmap patterns, for either auxiliary or main memory respectively, from the dhapix or dhmpix arrays.

But first we erase the previous color. Some of the pixels straddle main and auxiliary memory, and a couple of pixels don't. When we hit a pixel that straddles memory banks, we need to operate on a byte pair. This process may sound a little confusing but the code should make this a little clearer.

```
/* double hi-res page 1 plot */
void dhrplot(int x, int y, int drawcolor)
{
    int xoff, pattern;
    unsigned char *ptr;
/* bi-directional wrap supported for one scanline only for vector
line drawing */
    if (x < 0) x+= 140;
    if (x > 139) x-=140;
    if (x < 0 || x > 159 || y < 0 || y > 191)return;
    /* pixel position in 4 byte run */
    pattern = (x%7);

    /* starting byte in 2 byte run */
    xoff = (x/7) * 2;
```

```
/* first byte in main memory byte pair */
/* first byte in auxiliary memory byte pair */
ptr = (unsigned char *) (HB[y] + xoff);
switch(pattern)
{

    case 0: dhraux[0] = 0; /* select auxiliary memory */
            ptr[0] &= 0x70; /* 01110000 erase 4 bits */
            ptr[0] |= dhapix[drawcolor][0];
            /* 00001111 assign 4 bits */;
            dhrmain[0] = 0; /* reset to main memory */
            break;
    case 1: dhraux[0] = 0; /* select auxiliary memory */
            ptr[0] &= 0x0f; /* 00001111 erase 3 bits */
            ptr[0] |= dhapix[drawcolor][1];
            /* 01110000 assign 3 bits */
            dhrmain[0] = 0; /* reset to main memory */
            ptr[0] &= 0x7e; /* 01111110 erase 1 bit */
            ptr[0] |= dhmpix[drawcolor][1];
            /* 00000001 assign 1 bit */
            break;
    case 2: ptr[0] &= 0x61; /* 01100001 erase 4 bits */
            ptr[0] |= dhmpix[drawcolor][2];
            /* 00011110 assign 4 bits */
            break;
    case 3: ptr[0] &= 0x1f; /* 00011111 erase 2 bits */
            ptr[0] |= dhmpix[drawcolor][3];
            /* 01100000 assign 2 bits */
            dhraux[0] = 0; /* select auxiliary memory */
            /* advance offset in frame */
            /* second byte in auxiliary memory byte pair */
            ptr[1] &= 0x7c; /* 01111100 erase 2 bits */
            ptr[1] |= dhapix[drawcolor][3];
            /* 00000011 assign 2 bits */
            dhrmain[0] = 0; /* reset to main memory */
            break;
    case 4: dhraux[0] = 0; /* select auxiliary memory */
            /* second byte in auxiliary memory byte pair */
            ptr[1] &= 0x43; /* 01000011 erase 4 bits */
            ptr[1] |= dhapix[drawcolor][4];
            /* 00111100 assign 4 bits */
            dhrmain[0] = 0; /* reset to main memory */
            break;
    case 5: dhraux[0] = 0; /* select auxiliary memory */
            /* second byte in auxiliary memory byte pair */
            ptr[1] &= 0x3f; /* 00111111 erase 1 bit */
            ptr[1] |= dhapix[drawcolor][5];
            /* 01000000 assign 1 bit */
            dhrmain[0] = 0; /* reset to main memory */
            /* second byte in main memory byte pair */
            ptr[1] &= 0x78; /* 01111000 erase 3 bits */
```

```
                ptr[1] |= dhmpix[drawcolor][5];
                /* 00000111 assign 3 bits */
                break;
        case 6: /* second byte in main memory byte pair */
                ptr[1] &= 0x07; /* 00000111 erase 4 bits */
                ptr[1] |= dhmpix[drawcolor][6];
                /* 01111000 assign 4 bits */
                break;
    }

}
```

You will see in the code above the byte pair splits at meridian pixel 3 of 7 with the first half of the pixel in the high nibble of the first pair in the main memory bank and the second half of the pixel in the low nibble of the second pair in the auxiliary memory bank. Note also that we are following cc65's efficiency recommendation throughout this demo code to subscript pointers rather than increment them, so at the split we advance to subscript 1 to the second byte pair rather than increment the pointer itself.

Line Routines

The vertical line plotting function was discussed briefly above. But before we look at drawing vertical and horizontal lines in DHGR, let's talk about all the line plotting functions in this demo. The core line plotting function is a more or less standard line plotter fairly widely used. It does not work like the moveto() and lineto() type of

functions that are also widely used. There is no concept of a graphics cursor in this demo, except perhaps in the higher level line scripting function that draws a picture using a tables of x,y coordinates, and which calls the following function, which in turn calls the plotting function in the previous section:

```
/* Bresenham Algorithm line drawing routine for double hi-res
page 1 */
void dhrline(int x1, int y1, int x2, int y2, int drawcolor)
{
    int dx, dy, sx, sy, err, err2;
    if(x1 < x2) {
        dx = x2 - x1;
        sx = 1;
    }
    else {
        sx = -1;
        dx = x1 - x2;
    }
    if(y1 < y2) {
        sy = 1;
        dy = y2 - y1;
    }
    else {
        sy = -1;
        dy = y1 - y2;
    }
    err = dx-dy;
    for (;;) {
        dhrplot(x1,y1,drawcolor);
        if(x1 == x2 && y1 == y2)break;
        err2 = err*2;
        if(err2 > (0-dy)) {
             err = err - dy;
             x1 = x1 + sx;
        }
        if(err2 <  dx) {
             err = err + dx;
             y1 = y1 + sy;
        }
    }
}
```

There's not much else to say about the above. But for efficiency, 2 other line plotting functions are provided. One of these is the horizontal line plotting function, but it is really a block fill routine called dhrfbox() which draws a filled box to the color specified using double hi-res colors 0-15. Horizontal lines are filled boxes of one scan-line in height so we don't really need a horizontal line function.

The vertical line function also does double-duty in this demo. It is called by dhrfbox() discussed above and also by the font routine because the font routine is vertically scalable

so in the interests of efficiency of smaller code always uses a vertical line of either 1 or 2 pixels. The vertical line function therefore is designed to simply plot on its own without calling previous plotting function. The previously discussed plotting function could have been eliminated altogether, but for fun it was left in and also to show two slightly different table-driven methods of plotting pixels in DHGR. This one is more efficient in the fact that it uses a single pixel table, but takes a hit on unmasking the individual colors on the fly. Note also the use of the HB[] table to get the address of the scan-line:

```
/* double hi-res page 1 plot vertical line */
/* some duplicate code in here from dhrplot */
void dhrvline(int x, int y1, int y2, int drawcolor)
{
    int y, xoff, pattern;
    unsigned char *ptr;
    /* swap co-ordinates if out of order */
    if (y1 > y2) {
       y =  y2;
       y2 = y1;
    }
    else {
       y = y1;
    }
    if (x < 0 || x > 159 || y < 0 || y > 191)return;
    pattern = (x%7);
    if (pattern > 3)xoff = ((x/7) * 2) + 1;
    else xoff = (x/7) * 2;
    if (y2 > 191) y2 = 192;
    else y2++;
    while (y < y2) {
        ptr = (unsigned char *) (HB[y] + xoff);
        switch(pattern)
        {
            case 0: dhraux[0] = 0; /* select auxiliary memory */
                    ptr[0] &= 0x70;
                    ptr[0] |= (dhrbytes[drawcolor][0] &0x0f);
                    dhrmain[0] = 0; /* reset to main memory */
                    break;
            case 1: dhraux[0] = 0; /* select auxiliary memory */
                    ptr[0] &= 0x0f;
                    ptr[0] |= (dhrbytes[drawcolor][0] & 0x70);
                    dhrmain[0] = 0; /* reset to main memory */
                    ptr[0] &= 0x7e;
                    ptr[0] |= (dhrbytes[drawcolor][1] & 0x01);
                    break;
            case 2: ptr[0] &= 0x61;
                    ptr[0] |= (dhrbytes[drawcolor][1] & 0x1e);
                    break;
            case 3: ptr[0] &= 0x1f;
                    ptr[0] |= (dhrbytes[drawcolor][1] & 0x60);
                    dhraux[0] = 0; /* select auxiliary memory */
                    /* advance offset in frame */
```

```
                    ptr[1] &= 0x7c;
                    ptr[1] |= (dhrbytes[drawcolor][2] & 0x03);
                    dhrmain[0] = 0; /* reset to main memory */
                    break;
            case 4: dhraux[0] = 0; /* select auxiliary memory */
                    ptr[0] &= 0x43;
                    ptr[0] |= (dhrbytes[drawcolor][2] & 0x3c);
                    dhrmain[0] = 0; /* reset to main memory */
                    break;
            case 5: dhraux[0] = 0; /* select auxiliary memory */
                    ptr[0] &= 0x3f;
                    ptr[0] |= (dhrbytes[drawcolor][2] & 0x40);
                    dhrmain[0] = 0; /* reset to main memory */
                    ptr[0] &= 0x78;
                    ptr[0] |= (dhrbytes[drawcolor][3] & 0x07);
                    break;
            case 6: ptr[0] &= 0x07;
                    ptr[0] |= (dhrbytes[drawcolor][3] & 0x78);
                    break;
        }
        y++;
    }
}
```

Like any other function that uses the soft-switches to access auxiliary memory in this demo, the function above always returns with the soft-switch set to main memory. That way we always know where we are, and if we don't need auxiliary memory turned-on we save ourselves a little code; otherwise we would need to always switch to main memory explicitly everywhere just to be sure.

The comments in the dhrplot() function can be reviewed if you are somewhat confused about what is being done in the dhrvline() function shown above. As you can see, the dhrvline() function really is the dhrplot() function with a height argument. But it also uses the stack a little more, so at the end of the day a person might really be better off with dhrplot(). Evidently you have options. It is also fun to see how all this DHGR stuff works and that is the real purpose for this demo.

The next section is on graphics primitives. As noted previously, the dhrbox() function calls the dhrvline() function. As for efficiency, about all a person can do in DHGR mode is tweak the code here and there, because the problem still remains that isolating a pixel to plot a color in DHGR is nasty because it requires a double operation to erase the color that is there first, before plotting a new color.

A special purpose function which includes its own pixel level plotting like dhrvline() has is always an alternative if you want to reduce stack overhead at the expense of code-size. But the Apple II is not alone in needing what seems to be many steps to put a colored dot on the screen. However, DHGR is particularly as strange in its own way as the other Apple II color modes and as different as they all are from each other. It is only Monochrome DHGR that is almost as efficient as Monochrome HGR (Hi-Res).

Graphics Primitives

This demo has only two objects that could be described as graphics primitives; a circle and a box. For this demo only object outlines are supported, but the routines could easily be extended from current "wireframe" style line elements to provide fill attributes. For now these are sufficient.

The Circle

The circle is plotted using Bresenham's. It supports a horizontal aspect adjustment based on simple ratio over proportion, and technically it is really a half-scaled ellipse when used as a circle.

```
/* Bresenham Algorithm with crude aspect in the xterm only. 1 x 2
aspect for a circle. */
void dhrcircle(int x1, int y1, int radius, int drawcolor, int
xmult, int xdiv)
{
  int f = 1 - radius;
  int ddfx = 1;
  int ddfy = -2 * radius;
  int x = 0;
  int y = radius;
  int xxaspect, xyaspect;
```

```
  /* top and bottom */
  dhrplot(x1, y1 + radius, drawcolor);
  dhrplot(x1, y1 - radius, drawcolor);
  /* sides */
  /* a 1 x 2 aspect compresses the xterm for a circle */
  if (xmult > 0 && xdiv > xmult) xyaspect = (y * xmult)/xdiv;
  else xyaspect = y;
  dhrplot(x1 + xyaspect, y1, drawcolor);
  dhrplot(x1 - xyaspect, y1, drawcolor);

  while(x < y){
    if(f >= 0) {
      y--;
      ddfy += 2;
      f += ddfy;
    }
    x++;
    ddfx += 2;
    f += ddfx;
    if (xmult > 0 && xdiv > xmult) {
         /* a 1 x 2 aspect */
         /* compresses the xterm for a circle */
        xxaspect = (x * xmult)/xdiv;
        xyaspect = (y * xmult)/xdiv;
    }
    else {
        xxaspect = x;
        xyaspect = y;
    }
    /* top and bottom */
    dhrplot(x1 + xxaspect, y1 + y, drawcolor); /* bottom right */
    dhrplot(x1 - xxaspect, y1 + y, drawcolor); /* bottom left */
    dhrplot(x1 + xxaspect, y1 - y, drawcolor); /* top right */
    dhrplot(x1 - xxaspect, y1 - y, drawcolor); /* top left */
    /* sides */
    dhrplot(x1 + xyaspect, y1 + x, drawcolor); /* mid botright */
    dhrplot(x1 - xyaspect, y1 + x, drawcolor); /* mid botleft */
    dhrplot(x1 + xyaspect, y1 - x, drawcolor); /* mid topright */
    dhrplot(x1 - xyaspect, y1 - x, drawcolor); /* mid topleft */

  }

}
```

A filled disk routine based on the above might be pretty quick, and it is easy to code, but in reality how often do you need a filled disk? Half-Arcs are also easy too, but at the end of the day, how often do you even need a circle, let alone an arc? Be that as it may, the code above is there if you need it.

The Box

The Box likely needs no introduction. In previous discussion I have mentioned that the DHGR box routine in this demo just calls two other routines:

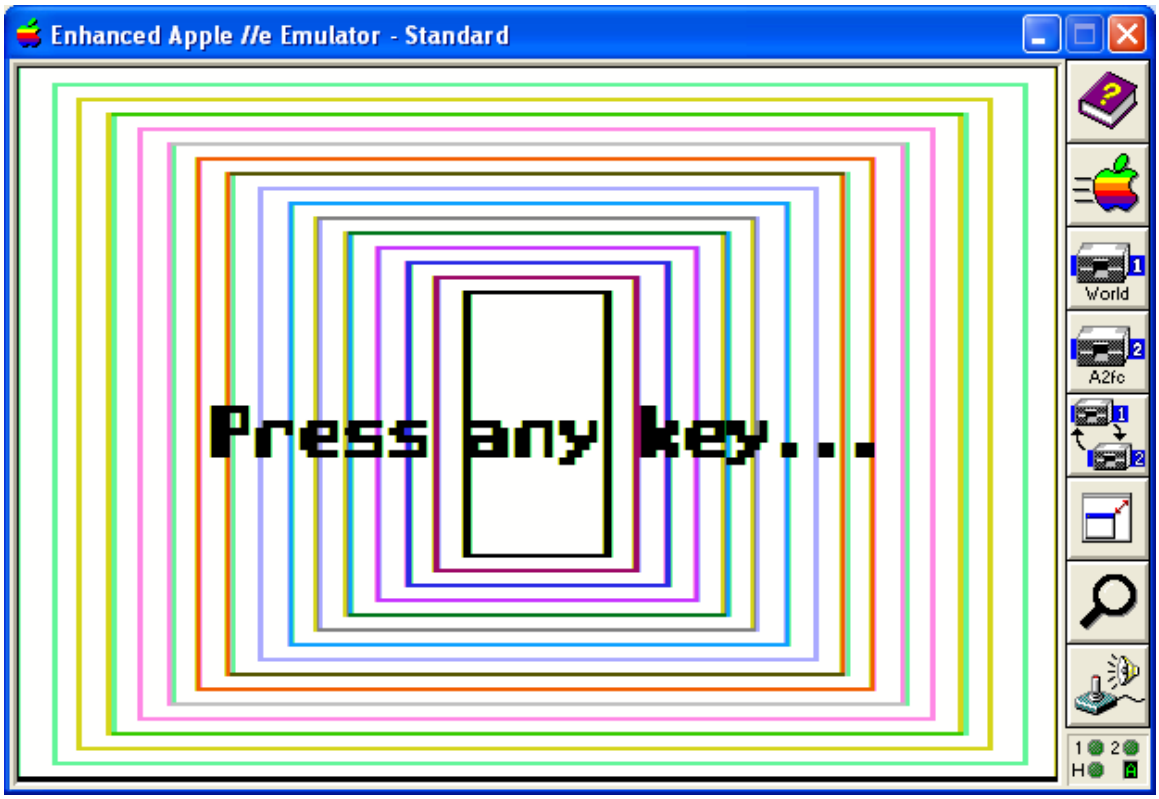
```
void dhrbox(int x1,int y1,int x2,int y2,int drawcolor)
{
     dhrfbox(x1,y1,x2,y1,drawcolor);
     y1++; y2--;
     dhrvline(x1,y1,y2,drawcolor);
     dhrvline(x2,y1,y2,drawcolor);
     y2++;
     dhrfbox(x1,y2,x2,y2,drawcolor);
}
```

Block Fill Routines

This demo does not have area fill routines. It provides 3 screen fill routines. One of them, the dhiresclear() routine, which was previously listed, just clears the DHGR screen to black. Without much work, it could also clear the screen to white, and that's about it.

The other two Block Fill routines are similar, and can be used to clear the screen to different colors, or to draw bands of color, or even horizontal lines (see The Box).

The reason we have two routines that are almost the same to draw filled boxes is that, besides being fun to write, a block aligned filled-box is much quicker in DHGR because it doesn't need to fill-in the orphaned pixels at either side, but sometimes you may want a pixel aligned block.



On the other hand if you want to reverse-video a menu selection with an xored box or just want a quick colored box and you have done your layout in 7 pixel increments, you are better-off to use block alignment and avoid the overhead of additional calculations to fill-in the sides.

```
/* Draws a filled box to the color specified using double hi-res
colors 0-16. Called in pixels but works only in 4 byte blocks. */
/* consider using dhrfbox instead of this routine unless you are
not doing other plotting and you want to do something like clear
the screen which does not require pixel level accuracy */
void dhrflood(int x1,int y1,int x2,int y2,int drawcolor)
{
    int x, packet, xorg, xend, idx, xoff;
    unsigned char *ptr, mainbuf[40], auxbuf[40];
    if (x1 < 0 || x2 > 139 || y1 < 0 || y2 > 191)return;
    /* convert pixels to 4 byte blocks */
    x = x1;
    while ((x%7)!=0) x ++; /* advance to left side of box */
    xorg = ((x / 7) * 4);
    idx = x2 + 1;
    xend = ((idx/7) * 4);
```

```
/* assign packet length */
idx = (xend-xorg);
if(idx < 4) return;
packet = idx / 2;
xoff = (xorg / 2);
if (drawcolor < 1 || drawcolor > 16) drawcolor = 0;

switch (drawcolor)
{
    case 16:
        /* pseudo-color 16 - inverse video */
        while(y1<y2)
        {
            ptr = (unsigned char *)(HB[y1] + xoff);
            dhraux[0] = 0; /* select auxiliary memory */
            memcpy(auxbuf,ptr,packet);
      for(idx=0;idx<packet;idx++)auxbuf[idx]^= (char) 0x7f;
            dhraux[0] = 0; /* select auxiliary memory */
            memcpy(ptr,auxbuf,packet);
            dhrmain[0] = 0; /* select main memory */
            memcpy(mainbuf,ptr,packet);
      for(idx=0;idx<packet;idx++) mainbuf[idx]^= (char) 0x7f;
            dhrmain[0] = 0; /* select main memory */
            memcpy(ptr,mainbuf,packet);
            y1++;
        }
        return;

    case 15: /* white or black set the buffer */
             memset(auxbuf,0x7f,packet);
             memset(mainbuf,0x7f,packet);
             break;
    case 0:  memset(auxbuf,0,packet);
             memset(mainbuf,0,packet);
             break;
    default:
        /* other colors */
        /* expand byte pairs to build scanline buffers */
        /* interleaf 7 pixels between main and aux memory */
        for (idx = 0; idx < packet; idx++) {
            auxbuf[idx]  = dhrbytes[drawcolor][0];
            mainbuf[idx] = dhrbytes[drawcolor][1];
            idx++;
            auxbuf[idx]  = dhrbytes[drawcolor][2];
            mainbuf[idx] = dhrbytes[drawcolor][3];
        }


}

/* now write the pixels */
```

```
   while(y1<y2)
   {
        ptr = (unsigned char *)(HB[y1] + xoff);
        dhraux[0] = 0; /* select auxiliary memory */
        memcpy(ptr,auxbuf,packet);
        dhrmain[0] = 0; /* select main memory */
        memcpy(ptr,mainbuf,packet);
        y1++;
    }

}
```

The function below is based on the block fill function above but provides pixel level fill at the expense of speed:

```
/* Draws a filled box to the color specified using double hi-res
colors 0-15. */
void dhrfbox(int x1,int y1,int x2,int y2,int drawcolor)
{
    int y, x, packet, xorg, xend, prefix, postfix, idx, xoff;
    unsigned char *ptr, mainbuf[40], auxbuf[40];
    if (x1 < 0 || x2 > 139 || y1 < 0 || y2 > 191)return;
    /* convert pixels to 4 byte blocks and
       calculate pixels before blocks */
    prefix = x1;
    /* advance to left side of box */
    while ((prefix%7)!=0) prefix++;
    xorg = ((prefix / 7) * 4);
    idx = x2 + 1;
    xend = ((idx/7) * 4);
    /* assign packet length */
    idx = (xend-xorg);
    /* if box width does not include a full 4 byte block
       draw a filled box using a series of vertical lines */
    if(idx < 4) {
       x2 += 1;
       for (x = x1; x < x2; x++)dhrvline(x,y1,y2,drawcolor);
       return;
    }
    packet = idx / 2;
    xoff = (xorg / 2);
   switch (drawcolor)
    {
        case 15: /* white or black set the buffer */
                 memset(auxbuf,0x7f,packet);
                 memset(mainbuf,0x7f,packet);
                 break;
        case 0:  memset(auxbuf,0,packet);
                 memset(mainbuf,0,packet);
                 break;
```

```
        default:
            /* other colors */
            /* expand byte pairs to build scanline buffers */
            /* interleaf 7 pixels between main and aux memory */
            for (idx = 0; idx < packet; idx++) {
                auxbuf[idx]  = dhrbytes[drawcolor][0];
                mainbuf[idx] = dhrbytes[drawcolor][1];
                idx++;
                auxbuf[idx]  = dhrbytes[drawcolor][2];
                mainbuf[idx] = dhrbytes[drawcolor][3];
            }
    }

   /* first write the horizontal pixel blocks */
   y = y1;
   y2++;
   while(y<y2)
   {
        ptr = (unsigned char *) (HB[y] + xoff);
        dhraux[0] = 0;
        memcpy(ptr,auxbuf,packet);
        dhrmain[0] = 0;
        memcpy(ptr,mainbuf,packet);
        y++;
    }
    y2--;
    /* now write the remaining pixels using a series of vertical
       lines expanding outward from the blocks */
    if (prefix != x1) {
       x1--;
       for (x = prefix;x > x1; x--) dhrvline(x,y1,y2,drawcolor);
    }

    postfix = ((xend / 4) * 7) - 1;
    if (postfix != x2) {
        x2++;
    for (x = postfix;x < x2; x++) dhrvline(x,y1,y2,drawcolor);
    }
}
```

Font Routines

The font routines in this demo are based on an Apple II bit-mapped font for HGR mode. We could easily have provided any number of fonts in an HGR format that would plot far more quickly and look nicer. There is memory under this demo program to load such a disk font. This is a demo, so I haven't bothered. This font is really a monochrome font and looks fine on the DHGR Monochrome Display and plots quickly in mono to 80 columns.

The pixels are pretty wide in DHGR so any font that is not specifically designed for display, with a 4 pixel bitmask for each letter can't be "blitted" right onto the DHGR color screen, so must plot pixel by pixel.

Having said that, the font routine in this demo does provide some features;

- proportional spacing
- double scaling in the vertical direction
- background color
- foreground color

```
/* Apple II 7 x 8 bitmapped font - can be used as-is in HGR mode
or as a character map for plotting a pixel-graphics font. This is
an extended and modified version of the __chr[760] array
originally shipped with the graphics library for the Aztec C65
CII compiler for DOS 3.3 from Manx Software Systems, 1983. I
modified and extended it in 1989-1990 in its present form (below)
to contain Ascii Values from 32-168 which includes multilingual
characters and symbols based on the IBM-PC extended character set
for MS-DOS. I further extended it to include all the IBM extended
characters in my AWINDLO utility in 2013. */

unsigned char __chr[1096] = {
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x0C, 0x0C, 0x0C, 0x0C, 0x0C, 0x00, 0x0C, 0x00,
...
0x3C, 0x66, 0x3C, 0x00, 0x7E, 0x00, 0x00, 0x00,
0x18, 0x00, 0x18, 0x0C, 0x06, 0x66, 0x3C, 0x00};

int msk7[]={0x1,0x2,0x4,0x8,0x10,0x20,0x40};

/* Double Hi-Res 16 color font routine */
void dhrfont(char *str,int row, int col, int fg, int bg,
int scale, int pitch, char justify)
{
int target, scanline, offset, r, r2, c, d, byte, nibble, x,
spaces = 0;
unsigned char ch;

  if (scale > 2)scale = 1;
  if (pitch != 6)pitch = 7;
  target = strlen(str);
  /* use a narrower width for plotting certain characters */
  for (byte=0;byte<target;byte++) {
        d = str[byte]&0x7f;
        if (d < 32)d = 32;
        if (d == 32 || d == 'l' || d == '1' ||
            d == '.' || d == '!')spaces +=1;
  }
```

```
  if (justify == 'm' || justify == 'M') {
      offset = ((target * pitch)- spaces)/2;
      col = col - offset;
  }

  for(scanline=0;scanline<8;scanline++) {
    /* set values for vertical term */
    /* expand x scale in the vertical direction */
    r = (scanline * scale) + row; /* max 16 high */
    if (r > 191)break;
    r2 = r + 1;
    /* run the string 8 times. if scale = 2 then print a double
       line each time which gives us a font of 16 high */
    spaces = 0;
    for (byte=0;byte<target;byte++) {
       /* calculate the starting column for each run
          in the width of 6 or 7 pixels */
      c = ((byte * pitch) + col)-spaces;
      if (c > 139)continue;
      d = str[byte]&0x7f;
      if (d < 32)d = 32;
      if (d == 32 || d == 'l' || d == '1'||
          d == '.'||d == '!')spaces +=1;
      if (d == 32 && bg < 0)continue;
      /* get the character bitmap for this line from the font */
      offset = ((d-32) * 8) + scanline;
      ch = __chr[offset];
      if (ch == 0 && bg < 0)continue;
      for (nibble=0;nibble<pitch;nibble++){
        x = c+nibble;
        if (x > 139)break;
        if (ch & msk7[nibble]){
             /* a foreground color always plots */
             if (scale > 1 && r2 < 192)
                 dhrvline(x,r,r2,fg);
             else
                 dhrvline(x,r,r,fg);
        }
        else {
             /* if not using a background color skip pixel(s) */
             if (bg < 0)continue;
             if (scale > 1 && r2 < 192)
                 dhrvline(x,r,r2,bg);
             else
                 dhrvline(x,r,r,bg);
          }
       }
    }
  }
}
```

Not much more can be said about this font routine. It calls dhrvline() as previously noted but could be modified to "blit" directly to screen memory much more efficiently using a DHGR native font with roughly the same logic and with a similar control structure and about the same amount of code.

Setting Video Modes

The following functions are used to set the video modes used in this demo. They are pretty self explanatory. Keep in mind that since we use the language card memory for other purposes in our linker config, we must use cc65's routine to safely set to 80 column mode, and not just something simpler that we write.

```
/* 80 column mode must be set before calling */
void dhireson(void)
{
    /* page 1 double hires */
    asm("sta $c050"); /* GRAPHICS                  */
    asm("sta $c052"); /* GRAPHICS ONLY, NOT MIXED */
    asm("sta $c054"); /* PAGE ONE                  */
    asm("sta $c057"); /* HI-RES                    */
    asm("sta $c05e"); /* TURN ON DOUBLE RES        */
}
```

The above is very similar to setting double lo-res mode because we are using page one in both cases.

```
/* the following two routines work with all Apple IIe graphics
modes */
void mixedtexton(void)
{
    asm("sta $c053"); /* MIXED TEXT/GRAPHICS */
}

void mixedtextoff(void)
{
    asm("sta  $c052"); /* GRAPHICS ONLY, NOT MIXED */
}

/* 80 column mode must be set to on after calling */
void dhiresoff(void)
{
    asm("sta $c051"); /* TEXT - HIDE GRAPHICS */
    asm("sta $c05f"); /* TURN OFF DOUBLE RES  */
    asm("sta $c054"); /* PAGE ONE             */
}
```

Mixed Text and Graphics

At the beginning of this document we "jumped in" to our pixel routines immediately after the tables because that was more fun. The text screen base address table was listed but we never had a chance to look at the text character routines. These same text character routines can be used in Double Lo-Res. So can the mode routines shown above. The idea here is to go right to text screen memory in 80 column mode. Remember to use cc65 to set to 80 column mode. Note also that these routines do not include the fix-up for reverse video and mouse-text characters. They are very simple like the rest of this:

```
/* some functions to output text directly to the screen in mixed
mode double lo-res and double hi-res graphics */
/* clear the bottom 4 lines of the text screen in mixed mode
double res */
void dloclear_bottom(void)
{
    char *crt;
    int row, col;
    char c = 32 + 128;
    for (row = 0; row < 4; row++) {
      crt = (char *)(dlotextbase[row]);
      for (col = 0; col < 40; col++) {
          dhraux[0] = 0; /* select auxiliary memory */
          crt[col] = c;
          dhrmain[0] = 0; /* reset to main memory */
          crt[col] = c;
      }
    }
}
/* print string directly to text screen memory in mixed mode
double res - row settings = 0,0 to 3,79 in mixed text and
graphics mode */
void dloprint_bottom(char *str,int row,int col)
{
     char *crt, c;
     int x, aux = 1, idx, jdx = 0;
     x = col / 2;
     if (col % 2) aux = 0;
     crt = (char *)(dlotextbase[row]+x);
     for (idx=0;;) {
       c = str[idx]; idx++;
       if (c == 0)break;
       c+=128;
       if (aux == 1) {
         dhraux[0] = 0; /* select auxiliary memory */
         crt[jdx] = c; aux = 0;
       }
       else {
         dhrmain[0] = 0; /* reset to main memory */
         crt[jdx] = c;
```

```
            jdx++; aux = 1;
         }
      }
      /* safety play */
      dhrmain[0] = 0; /* reset to main memory */
}
```

Building the Demo

This concludes the core routines used by this DHGR demo. They are stored in world.h for convenience and to simply keep them together so world.c can be compiled without the need for anything special, but in the end these core routines will likely have their own library.

The Make File for this demo works with gcc make. It's as convenient as the rest of it:

```
PRG=world
#put your own path to the cc65 working directory here
CC65ROOT=c:\cc65_snap

$(PRG).PRG: $(PRG).c
      cl65 -O -t apple2enh -C $(PRG).cfg $(PRG).c
      del $(PRG).o
```

Existing users of cc65 will likely already know what to do with this if anything.

To build all this, Windows users can get the latest cc65 snapshot, and then create a path similar to the following:

```
C:\cc65_snap\PROGRAMS\DHGR\
```

And set-up a batch-file similar to the following:

```
C:\cc65_snap\cc65-env.cmd

@echo off
SET CROOT=C:\cc65_snap
SET PATH=%CROOT%\bin;%CROOT%\tools;%PATH%
SET CA65_INC=%CROOT%\asminc
SET CC65_INC=%CROOT%\include
SET LD65_CFG=%CROOT%\cfg
SET LD65_LIB=%CROOT%\lib
SET LD65_OBJ=%CROOT%\obj
```

A Windows shortcut is provided in the distribution at the following link:

http://www.appleoldies.ca/cc65/programs/dhgr/dhiworld.zip

Unzip to: `C:\cc65_snap\PROGRAMS\DHGR\` (as noted above) or similar.

The WORLD Program

```
/* ------------------------------------------------------------
System        : PRODOS 8
Platform      : Apple IIe Enhanced 128K or equivalent
Program       : world.c
Description   : Demo Program
                Pixel and Line Graphics and Color Fonts using
                Double Hi-Res 140 x 192 x 16 color mode.
                Mixed-Mode double-res text screen routines.
Written by    : Bill Buckels
Date Written  : January 2013
Revision      : 1.0 First Release - Manx Aztec C65 Version 3.2b
                MS-DOS cross-development environment
Date Revised  : May 2014
                2.0 Second Release - cc65
                cross-development environment for current cc65
snapshot
                includes Windows, Linux, and Mac OSX
Licence       : You may use this program for whatever you wish as
long
                as you agree that Bill Buckels has no warranty or
                liability obligations whatsoever from said use.
------------------------------------------------------------ */

#ifndef __APPLE2__
#define __APPLE2__
#endif
#include <string.h>
#include <conio.h>
#include "world.h"
#define WORLDMAP 0
#define USAMAP 1
#define MAPLELEAF 2

/* the coordinates to draw the whole world */
int WORLD[] = {...};
/* the coordinates to draw the USA */
int USA[] = {...};
/* the coordinates to draw the canadian flag */
int MAPLE[] = {
  0, 10, 15, 10, 15,180,  0,180,  0, 10, -1, -1,
124, 10,139, 10,139,180,124,180,124, 10, -1, -1,
 32, 62, 43, 67, 46, 53, 57, 71, 53, 32, 62, 40, 70, 11, 78, 40, 86, 32,
 83, 71, 93, 53, 97, 67,108, 62,105, 91,110, 98, 91,133, 95,151, 72,145,
 72,180, 68,180, 68,145, 45,151, 49,133, 29, 98, 34, 91, 32, 62, -1, -2};
```

The drawing script arrays above are not entirely listed to save space in this document. Each array consists of sections of line elements of positive x,y coordinate values. Each section is terminated with x,y values of -1. The end of the array is terminated with a y value of -2. These are similar to Apple II shape tables in a way, and in a way not.

The following array rotates through colors for the USA Map portion of the demo. These colors are simply the DHGR colors that seemed to look good when plotting the states:

```
int colors[] = {
LOLTBLUE,LOPINK,LOLTGREEN,LOYELLOW,LOAQUA,LOWHITE,-1};
```

The following function draws the vector objects in the line table arrays. It is self-explanatory.

```
void drawmap(int mapidx,int drawcolor)
{
  int *ptr, xidx=0,yidx=1,newx,oldx,newy,oldy,color,cidx=0;
  /* select vector map or vector object */
  switch(mapidx)
  {
      case MAPLELEAF:   ptr = (int *)&MAPLE[0];
                        break;
      case USAMAP:      ptr = (int *)&USA[0];
                        break;
      case WORLDMAP:
      default:          ptr = (int *)&WORLD[0];
  }
  /* draw selected map or object */
  for (;;) {
      /* alternating color option */
      if (drawcolor < 0 || drawcolor > 15) {
            if (colors[cidx] < 0)cidx = 1;
            color = colors[cidx];
            cidx++;
      }
      else {
          color = drawcolor;
      }
      oldx= ptr[xidx];
      oldy= ptr[yidx];
      for(;;)
      {
          xidx +=2; yidx +=2;
          newx= ptr[xidx];
          newy= ptr[yidx];
          if (newx == -1) break;
          /* draw next line */
          dhrline(oldx,oldy,newx,newy,color);
          oldx=newx;
          oldy=newy;
       }
       if (newy == -2)break;
       xidx +=2;
       yidx +=2;
   }
}
```

The Main Program

So now that you have seen what went into writing the core DHGR pixel graphics and other DHGR routines, and the line drawing script routines for world.c, let's take a look at the main() program. The comments are self-explanatory. Reviewing the various functions earlier in this document will likely help you make sense of all of this. This is a very simple demo. But it covers the basics of what you need to do to write pixel graphics functions for the Apple II's DHGR mode.

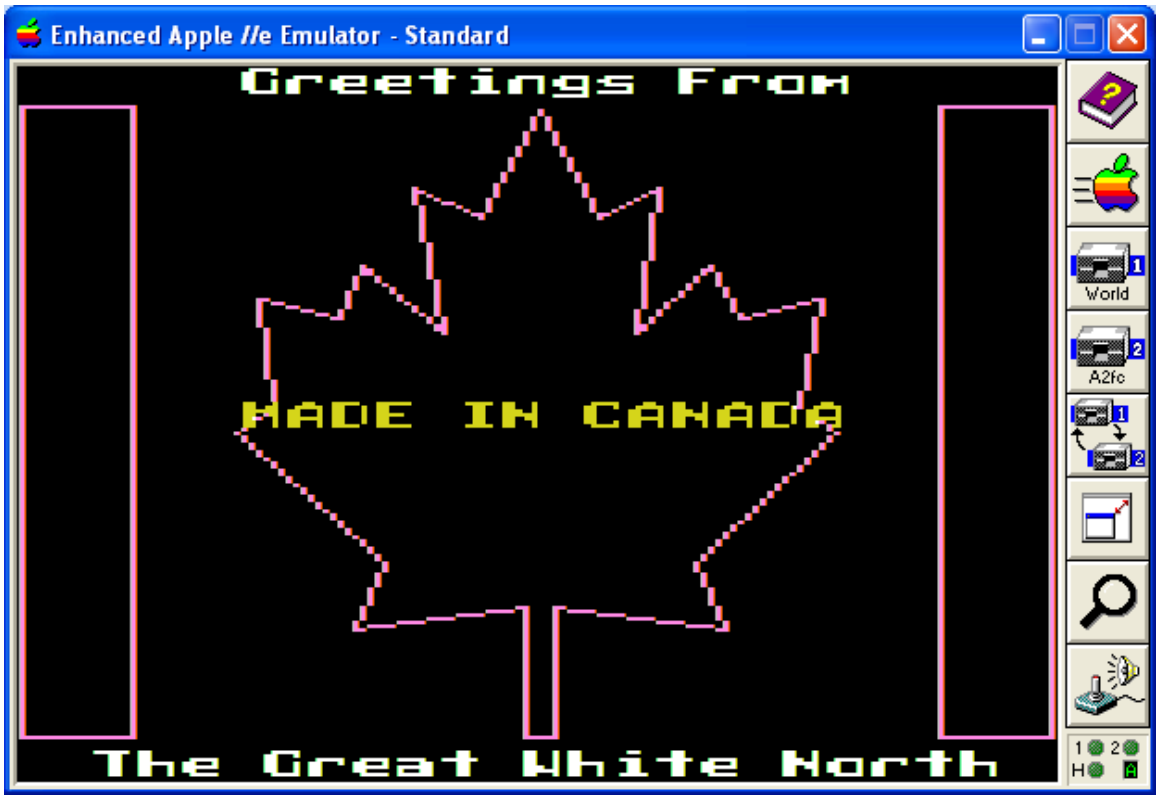
If you have run the demo already you already know how lame it is in terms of the potential of the DHGR display. But it is intended to be lame, and with a little imagination anyone can probably do better… and that's the idea here. We're just taking a look at this and first we need to understand how to make DHGR fit into cc65.

Some issues with this demo have been "sluffed" altogether. Notably, there are many inefficiencies in control structures which have been kept fairly verbose for clarity. The circle function could be tweaked to avoid "jaggies" and the reverse video could be quicker. Making improvements is left to the reader. They are likely endless.

```
int main(void)
{
    int idx,c,x2,y2;
    /* initialize 80 column card */
    /* this inializes the language card safely for cc65's
       standard memory configuration */
    videomode(VIDEOMODE_80COL);
    clrscr();
    /* turn-on double hi-res */
    dhireson();
    /* dline.c box demo */
    dhiresclear();
    for (idx = 1; idx < 16; idx++) {
        c = (idx * 4);
        x2 = 139-c;
        y2 = 191-c;
        dhrbox(c,c,x2,y2,idx);
    }
    dhrfont("Press any key...",91,70,15,-1,2,6,'M');
    cgetc();
    dhrflood(0,0,139,191,16);
    cgetc();
    /* circle demo */
    dhiresclear();
    for (idx = 1; idx < 16; idx++) {
        c = (idx * 4);
        dhrcircle(70, 96, 96-c, idx, 1, 2);
    }
    dhrfont("Press any key...",91,70,15,-1,2,6,'M');
```

```
    cgetc();
    dhrflood(0,0,139,191,16);
    cgetc();
    /* first screen */
    dhiresclear();
    dhrfont("DHGR Graphics in cc65",0,70,LOAQUA,-1,1,6,'M');
    dhrfont("A Demo by Bill Buckels",184,70,LOYELLOW,-1,1,6,'M');
    drawmap(USAMAP,-1);
    cgetc();
    /* second screen */
    dhiresclear();
    dhrcircle(70, 96, 60, LOLTBLUE,-1,-1);
    dhrfont("Press a Key to C",0,70,LOAQUA,-1,1,6,'M');
    dhrfont("The Wonderful World",80,70,LOPINK,-1,2,6,'M');
    dhrfont("Of DHGR Graphics",96,70,LOPINK,-1,2,6,'M');
    dloclear_bottom();
    mixedtexton();
    dloprint_bottom("Now that you've seen the USA",1,0);
    dloprint_bottom("Press a Key to See the World!",3,0);
    cgetc();
    /* third screen */
    mixedtextoff();
    dhiresclear();
    drawmap(WORLDMAP,LOPINK);
    cgetc();
    /* fourth screen */
    dhiresclear();
    drawmap(MAPLELEAF,LOPINK);
    dhrfont("Greetings From",0,70,LOWHITE,-1,1,6,'M');
    dhrfont("The Great White North",184,70,LOWHITE,-1,1,6,'M');
    dhrfont("MADE IN CANADA",90,70,LOYELLOW,-1,1,6,'M');
    cgetc();
    dhrflood(0,0,139,191,16);
    cgetc();
    /* clear double hi-res screen for the last time */
    /* then turn-off double hi-res */
    dhiresclear();
    dhiresoff();
    /* set to 80 column again so cc65 can exit cleanly */
    videomode(VIDEOMODE_80COL);
    clrscr();
    return 0;
}
```

Additional Notes



Bitmapped DHGR demos are sexier than pixel graphics demos like this one. There is no doubt about that.

References

This demo was written almost in its entirety using the following reference:

http://apple2.boldt.ca/?page=til/tn.aiie.003

Bill Buckels
bbuckels@mts.net