Introduction to Super Hi-Res in cc65

Chapter Four - Super Hi-Res Sprites and Image Fragments in cc65

Table of Contents

Introduction to Super Hi-Res in cc65.....1
Chapter Four - Super Hi-Res Sprites and Image Fragments in cc65.....1
Table of Contents.....1
Forward.....3
    Cc65 Overview.....3
    Help is on the Way.....3
        Demos and Documents to Date.....4
        Utilities To Date.....4
    Overall Approach.....4
    Tools, Tools, and More Tools.....5
    About This Document.....5
Introduction.....6
Super Hi-Res in cc65 until Now.....7
Super Hi-Res in cc65 Now.....7
A New Apple II File Format for Image Fragments.....7

BMP2Sprite (b2sprite) – Writing an SHR File Converter....9
Matching 24-Bit Colors to SHR 16-Color Palettes....10
Getting a 16-Color SHR Palette to Match-To....11
Initializing an SHR 16-color Palette....12
Double Precision Pre-Calculated Comparison Tables....12
Character Array Comparison Tables....12
Initializing a Built-In Palette....13
Initializing a Palette from an Existing SHR File....14
Converting a 24-bit BMP to an SHR Image Fragment....15
The main() Program and Tying it Together....19
Command Line Options....21
Selecting Output Format Options....21
Defaults....21
Flexible Output Format....21
Background Color Option "-b"....22
No Header Option "-nh"....22
No Palette Option "-np"....22
Selecting Input Options....22
Palette Option "-p"....22
Variations of Option "-p"....22
"PIC" File Palette Over-Ride Cancels Option "-p"....23
The Bounce Demo – Embedding an Image Fragment 23
Bouncing a Sprite....24
Putting a Sprite on the SHR Screen....26
The Bounce Demo main() Program 26
The Fraglode Demo - Loading an Image Fragment....28
Displaying an Image Fragment File....29
Making an SHR File List....31
The Fraglode Demo main() Program....32
Cc65 SHR Core Routines....33
Auxiliary Memory Routines....33
Setting-up the SHR Display....33
SHR Soft Switches....33
SHR Initialization....34
SHR Initialization for Bounce Demo....34
SHR Initialization for Fraglode Demo....35
Clearing the SHR Screen for Bounce Demo....37
SHR Palettes for Bounce Demo....37
Selecting 16 Colors....37
Setting an SHR Palette....38
Building the cc65 Demo Programs....38
Download These Projects....38
Additional Notes....39
How I Got Into This Mess....39

Forward

Cc65 Overview

When it comes to **writing cc65 programs for the the Apple II** , **Cc65** is a capable and modern C 6502 8 bit cross-compiler. A C language programmer can use cc65 to build Apple II programs from the comfort of a modern operating system including **Mac OS X**

Windows users can easily set-up cc65 by downloading the latest cc65 **Windows "Snapshot"** and unzipping it into a directory (i.e. "cc65_snap").

For additional information about cc65 start with the following link:

http://cc65.github.io/cc65/

Wikipedia says of cc65 that "The C library is quite extensive, and allows extensive usage of the target platform's hardware." While certainly true and a phenomenal body of work by cc65's Apple II developers like **Oliver Schmidt** has gone into the Apple II's platform specific cc65 libraries, the Apple II is more extensive than cc65's extensive Apple II C library and the Apple II's extensive hardware is more extensive than cc65's extensive usage of the Apple II's hardware, so (as with any C compiler) additional programming is required to do anything outside the functionality supported by the C libraries bundled with the compiler.

When it comes to Apple II graphics, only two Apple II graphics modes are currently available through cc65's tgi graphics driver support:

- Lo-Res Graphics Mode (LGR)
- Hi-Res Graphics Mode (HGR)

And when it comes to Apple II Hardware, "There's currently no support for direct hardware access. This does not mean you cannot do it, it just means that there's no help."

Help is on the Way

After procrastinating for a number of years, in May 2014 I decided to write and distribute a series of cc65 Apple II programs complete with source code and documentation targeted at gaps in cc65's support for the Apple II. I started with the missing Apple II Graphics Modes:

- Double Lo-Res (DLGR)
- Double Hi-Res (DHGR)
- Super Hi-Res (SHR)

The following documents and demos comprise some of what I have done so far:

Demos and Documents to Date

SHR	**Bit-Mapped Graphics**
Demo	**Image Loader: http://www.appleoldies.ca/cc65/programs/shr/piclode.zip**
Demo	**SlideShow: http://www.appleoldies.ca/cc65/programs/shr/picshow.zip**
SHR	**Pixel Graphics**
Demo **Doc**	**http://www.appleoldies.ca/cc65/programs/shr/shrworld.zip** **http://www.appleoldies.ca/cc65/docs/shr/shrworld.pdf**
DLGR	**Bit-Mapped Graphics**
Demo	**http://www.appleoldies.ca/cc65/programs/dlgr/dloshow.zip**
Doc	**http://www.appleoldies.ca/cc65/docs/dlgr/dloshow.pdf**
DLGR	**Pixel Graphics**
Demo	**http://www.appleoldies.ca/cc65/programs/dlgr/dlodemo.zip**
Doc	**http://www.appleoldies.ca/cc65/docs/dlgr/dlodemo.pdf**
DHGR	**Bit-Mapped Graphics**
Demo	**http://www.appleoldies.ca/cc65/programs/dhgr/dhishow.zip**
Doc	**http://www.appleoldies.ca/cc65/docs/dhgr/dhishow.pdf**
DHGR	**Pixel Graphics**
Demo	**http://www.appleoldies.ca/cc65/programs/dhgr/dhiworld.zip**
Doc	**http://www.appleoldies.ca/cc65/docs/dhgr/dhiworld.pdf**
Other	**Miscellaneous (not Graphics)**
Demo	**No-Slot Clock:** **http://www.appleoldies.ca/cc65/programs/REALTIME.zip**

Utilities To Date

http://www.appleoldies.ca/graphics/index.htm

These converters were written for the Aztec C65 compiler and not cc65.

Overall Approach

My view of cc65 is different from a compiler developer's view. When it comes to the improving the cc65 compiler's existing toolset itself, and marshalling my code into cc65's existing libraries; I have little immediate interest nor time to do so.

My view is one of an Apple II enthusiast and retro-programmer exploring the Apple II with the cc65 compiler and writing programs along the way. My documentation is both narrative and technical, so the individual documents, including this one, are more like blog-pieces with several rants thrown-in for good measure.

They are more than anything for the average "all-weather" C programmer like me who doesn't always depend on a built-in toolbox to get the job done, and who enjoys doing "knock-off" applications in low-level C with a limited smattering of 6502 assembler.

Tools, Tools, and More Tools

Since I started programming I have been fascinated with writing utilities. My view of C compilers is a little "old fashioned". Early C compilers always came with lots of utilities and examples, but somehow they were never enough, and from that day to this I have maintained the view that utilities and their development is very much a part of both compilers and the programmers who use them.

I am therefore more than somewhat at odds with the narrower scope of the compiler developer who does not share my "everything including the kitchen sink" perspective that there should be no limit to the utilities (and documentation and demos) that are available as part of a compiler's toolset.

About This Document

This document is even more like some of my older utilities documentation than what I have done recently with my cc65 documents (which are also quite narrative), in that I discuss the target platform and the utility in the "same breath" and in some respects I am quite irreverent.

But more importantly, with this document, now that I have got the basics of SHR graphics in cc65 "knocked off" in recent demos and documentation, I introduce the idea of writing your own tools and utilities (in this case a graphics converter) and using the product of your own utilities in developing routines and test programs for those routines.

On the local level, this document is yet another "recipe" in the cc65 Apple II SHR "cookbook", but on the global level it seems to me that providing example methodology for an average programmer like me to use SHR "Sprites" and image fragments in a cc65 program adds a great deal more value to the cc65 compiler than passively restricting functionality in favor of scope.

Having said that, the same idea holds true of any development environment and my view is simply a different point of view.

Bill Buckels
bbuckels@mts.net
July 2014

Introduction

This document is about 3 programs:

- B2Sprite – Converts Windows BMP's to Apple II Super Hi-Res Image Fragments
- Bounce – Demo for Embedded Image Fragments created by B2Sprite
- Fraglode – Demo for Image Fragment Files created by B2Sprite

The B2Sprite file converter is written in Ansi C. It is distributed with source code, a Win32 Binary compiled under MinGW, and a gcc compatible MAKEFILE so users of Linux and other platforms can compile their own binaries and follow along.

The two demo programs are written for the cc65 6502 C compiler in ISO C. They are instructional demos for programmers and part of a larger collection of documentation and demos for writing cc65 Apple II programs that take advantage of the Apple II's features that are not directly supported by cc65's link libraries.

The cc65 compiler comes with direct tgi driver support for only 2 of the Apple II's graphics modes; HGR and LGR. The Apple II's Double-Res Modes (DHGR and DLGR)

and Super Hi-Res Mode (SHR) that are not directly supported by cc65's libraries are already supported within this larger collection. This collection is a work in progress and will take a considerable amount of time to complete, but the programs in this document are complete to a functional point, so like the rest of this collection are being released as they are finished but are subject to updates and improvements.

Before reading this document you may wish to review the previous Apple II graphics documentation and demos previously listed and already in this collection.

Super Hi-Res in cc65 until Now

Until now, the SHR documents in this collection have focused on writing Super Hi-Res programs in cc65, and not really concerned themselves about where SHR bit-mapped graphics come from, or about other uses for them besides viewing them.

If you want to do more with Super Hi-Res graphics you will need to acquire them somehow, and get them onto the Apple II in some usable format. It is also obvious that graphics image fragments that are smaller than the Apple II display are needed for writing games and other similar Apple II programs.

We have not looked at doing anything with SHR image fragments in this collection until now.

Super Hi-Res in cc65 Now

Now that we have covered the basics of SHR programming in cc65, we can get more ambitious and set-out to write ourselves an SHR graphics converter that makes both SHR screen images and SHR image fragments, and two cc65 programs to test these in . Of course you are expected to expand on all of this a great deal if you wish to do more, and future documents and demos will also show you how to do more with this.

Since the Apple II does not have Sprite Registers like the Commodore 64, there is no practical difference between a Sprite and an Image Fragment on the Apple II, except that a Sprite needs to move around the screen so the screen palettes and the sprite palettes need to match on all areas of the SHR screen where the Sprite will perform.

This means that the simplest way to match background screens and sprites is to use only a single SHR palette of 16 colors. Image fragments don't move around much but to keep things simple in our demos and our converter, we are simply sticking to a single SHR palette of 16 colors.

A New Apple II File Format for Image Fragments

We also need a simple image format that doesn't take a lot of programming. So we will invent one that meets our present needs and some of our anticipated future needs…

Technically Unsupported

File Type: $C1 (193)
Auxiliary Type: $CC65

Full Name: UnPacked Apple IIgs Super Hi-Res 16 Color Screen Image Fragment File
Short Name: UnPacked Super Hi-Res Image Fragment
Written by: Bill Buckels August 1, 2014

Files of this type and auxiliary type contain an unpacked Apple IIGS Super Hi-Res 16 color screen image fragment with or without Header, ColorTable and Background Color.

Files of type $C1 and auxiliary type $CC65 contain an unpacked Apple IIGS Super Hi-Res 16 color screen image fragment, with an optional 2 byte header, followed by an optional single ColorTable, followed by an optional 1 byte background color, followed by pixelData. These files are in the range of 2 to 320 pixels x 1 to 200 scan lines. Pixels are stored in pairs to support byte aligned loading to the Super Hi-Res screen area.

Definition

Color table	A table of sixteen two-byte entries, where each entry in the table is a master color value ($0RGB, where R is the red component, G is the green component and B is the blue component).
File Structure	The format for these files is smaller than mode320 screen images.
Header	(+000) 1 Byte – width value in bytes
(optional)	(+001) 1 Byte - height value in scan lines.
colorTable (optional)	(+003) 32 Bytes - One Color table for all scan lines. When a Color table is not present the loader is expected to interpret a ColorTable, either from a background screen image or by some other method.
Background	1 Byte – Background (transparent) color 0-15
(optional)	When background color is disabled this byte is set to $FF (255)
pixelData	Width in bytes x Height in scan lines Bytes Pixel data to be displayed on the Super Hi-Res screen. Scan lines are from top to bottom.

Files of type $C1 and auxiliary type $CC65 can also be referred to as "sprite files" and are designed to be as flexible as possible. Loaders can read the optional headers of these files to interpret the optional information by doing a file size calculation based on the size of the Pixel data. If the file size is 2 bytes over the size of the Pixel data, no optional information is present and must be interpreted. If the file size is 34 bytes over the size of the Pixel data, the file contains a Color table but has no background color. If the file size is 35 bytes over the size of the Pixel Data the file contains all information fields.

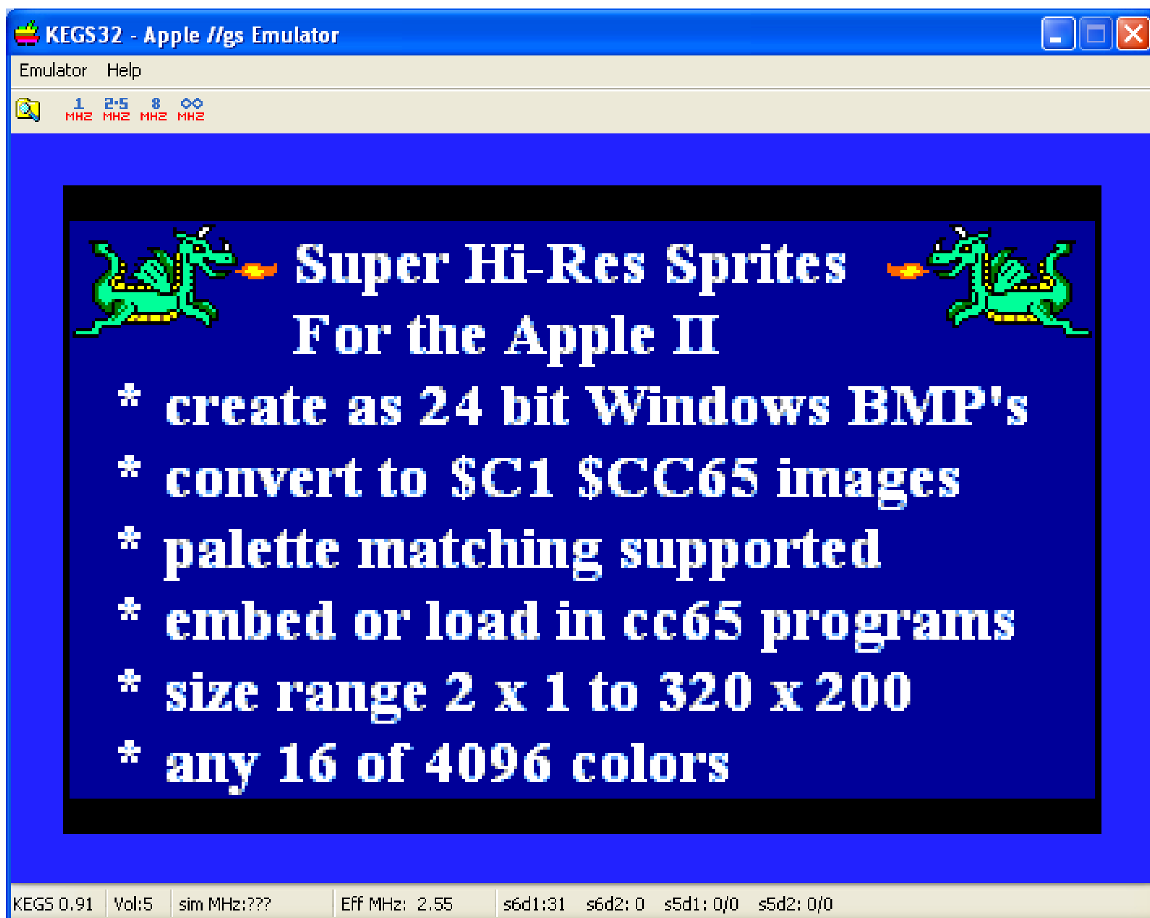
Further Reference

- Apple II File Type Notes, File Type $C1, Auxiliary Type $0000

You can see in our specification for our new format above that some fields are optional. For today's purposes we won't concern ourselves with much more rationale than the spec above, and some common-sense will decide how far we will go with all of this. But if we are going to the trouble to write a converter for our new format, we will make sure that our converter provides all the options required by our spec.

Some folks may argue that we don't need a new format since existing formats like the APF format allow for image fragments, or that RAW data is too bulky and existing formats like APF are compressed and take less disk space, but today we are keeping it simple so we learn something. Later on, we may even keep it simple so we do something easily and quickly that we can understand easily.

BMP2Sprite (b2sprite) – Writing an SHR File Converter



Now that we have an SHR file format established we will set-out to write a simple command-line SHR file converter in Ansi C. We use the 24-bit BMP format as our input format because it is easily created in Windows Paint and other programs, and is widely pasted and used. Converting a jpeg or a png or a gif to a BMP is simply a matter of saving to a BMP, so files from the Internet can easily be used as graphics sources.

From a programming view, writing a BMP reader for our converter is really simple. So is matching our colors to a 16 color palette. But we will need to do a little research to determine the best way to match our colors as closely as possible to a 16 color palette. We could also use dithering but today we are keeping it simple. We will only use pure-color matching using a color distance technique based-on visual perception rather than earlier techniques like simple reduction methods used in old-style conversion programs.

Matching 24-Bit Colors to SHR 16-Color Palettes

The technique we will use in our converter to match our SHR palettes to the 24-bit colors in our source BMP's is in the following excellent article:

Joel Yliluoma's arbitrary-palette positional dithering algorithm:

http://bisqwit.iki.fi/story/howto/dither/jy/

The basics including the formula we use are in the **GetDrawColor()** function listed below. This function "does it all" and simply returns the color index value to the closest match in a 16 color SHR palette when it is provided with the 24-bit rgb gun values used by our BMP input file to store its pixels!

```
/* use CCIR 601 luminosity to get closest color in SHR palette */
uchar GetDrawColor(uchar r, uchar g, uchar b)
{
    uchar drawcolor, red, green, blue;
    double dr, dg, db, diffR, diffG, diffB,
           luma, lumadiff, distance, prevdistance;
    int i;

    red = (uchar)(r >> 4);
    green = (uchar)(g >> 4);
    blue = (uchar)(b >> 4);

    /* quick check for verbatim match */
    for (i = 0; i < 16; i++) {
        if (rgbAppleArray[i][0] == red &&
            rgbAppleArray[i][1] == green &&
            rgbAppleArray[i][2] == blue) return (uchar)i;

    }

    dr = (double)r;
    dg = (double)g;
    db = (double)b;
    luma = (dr*299 + dg*587 + db*114) / (255.0*1000);
    lumadiff = rgbLuma[0]-luma;

    /* Compare the difference of RGB values,
       weigh by CCIR 601 luminosity */
    /* set palette index to SHR color with shortest distance */
    /* get color distance to first SHR palette color */
    diffR = (rgbDouble[i][0]-dr)/255.0;
```

```
    diffG = (rgbDouble[i][1]-dg)/255.0;
    diffB = (rgbDouble[i][2]-db)/255.0;
    prevdistance = (diffR*diffR*0.299 + diffG*diffG*0.587 +
                    diffB*diffB*0.114)*0.75 +
                    lumadiff*lumadiff;
    /* set palette index to first SHR color */
    drawcolor = 0;

    /* get color distance to rest of SHR palette colors */
    for (i=1;i<16;i++) {
        /* get color distance of to this SHR index */
        lumadiff = rgbLuma[i]-luma;
        diffR = (rgbDouble[i][0]-dr)/255.0;
        diffG = (rgbDouble[i][1]-dg)/255.0;
        diffB = (rgbDouble[i][2]-db)/255.0;
        distance = (diffR*diffR*0.299 + diffG*diffG*0.587 +
                    diffB*diffB*0.114)*0.75 +
                    lumadiff*lumadiff;
        /* if distance is smaller use this index */
        if (distance < prevdistance) {
           prevdistance = distance;
           drawcolor = (uchar)i;
        }

    }
    return drawcolor;
}
```

As you can see above, we are simply going through a 16 color palette and returning the closest match based on some kind of **Luma distance**. So where did this 16 color palette come from?

Getting a 16-Color SHR Palette to Match-To

It is obvious that in order to match to an SHR palette, we need to build a known palette to match to. We therefore will want some options:

- Built-In Known Palettes of 16-colors
- Palettes Extracted From Existing SHR Background Images

That was simple! In previous documents we have been using variations of the Apple II Lo-Res Colors also used by DHGR and DLGR so we continue to use those 4 palettes as our built-in palettes. We could be even fancier and accept popular palette formats to match to, but we are keeping things simple for now. But as a bonus we are learning code that can be re-used to write converters for the other Apple II Graphics modes later on!

While it is convenient for us to extract a palette from an existing SHR image, since we are keeping it simple, we will only extract palette 0 from an existing SHR image. Let’s first see how we construct the palettes used by **GetDrawColor()** and then take a brief look at how we get the values for those colors.

Initializing an SHR 16-color Palette

In **GetDrawColor()** several arrays are used. When our converter starts-up it need to decide where the SHR palette will come-from, and then create the tables for color matching of our BMP source colors.

Double Precision Pre-Calculated Comparison Tables

The code below takes a 16 color rgb palette character array and precalculates the double precision values used by the color distance comparison in **GetDrawColor()**. This speeds-up comparison to every pixel in our BMP, so cuts-down on processing time.

```
double rgbLuma[16], rgbDouble[16][3];

/* intialize the values for the current palette */
void InitDoubleArrays()
{
    int i;
    double dr, dg, db;
    unsigned r, g, b;

    /* array for matching closest sprite color in SHR palette */
    for (i=0;i<16;i++) {
        rgbDouble[i][0] = dr = (double) rgbArray[i][0];
        rgbDouble[i][1] = dg = (double) rgbArray[i][1];
        rgbDouble[i][2] = db = (double) rgbArray[i][2];
        rgbLuma[i] = (dr*299 + dg*587 + db*114) / (255.0*1000);
    }

}
```

Character Array Comparison Tables

In the **InitDoubleArrays()** function above and in **GetDrawColor()** function above that, we get rgb values for our SHR palette from character arrays:

```
uchar rgbArray[16][3], rgbAppleArray[16][3];
```

The values in **rgbArray** are in the BMP's 24-bit gun value format and the values in **rgbAppleArray** are in the SHR's 12-bit gun value format. But they are equivalent values. If we are using a built-in palette these arrays are initialized using the **GetBuiltinPalette()** function, and if we are using the palette from an existing SHR file, these arrays are initialized using the **GetSHRPalette()** function. The code for both of these follows. When we get to our **main() program** code we'll see how we decide in practice which one of these to use.

Initializing a Built-In Palette

```
/* use built-in palette if no shrfile is named */
sshort GetBuiltinPalette(sshort palidx)
{
    sshort i,j;
    uchar r,g,b;

    switch(palidx) {
        case 3: for (i=0;i<16;i++) {
                    rgbArray[i][0] = awinnewcolors[i][0];
                    rgbArray[i][1] = awinnewcolors[i][1];
                    rgbArray[i][2] = awinnewcolors[i][2];
                }
                break;
        case 2: for (i=0;i<16;i++) {
                    rgbArray[i][0] = awinoldcolors[i][0];
                    rgbArray[i][1] = awinoldcolors[i][1];
                    rgbArray[i][2] = awinoldcolors[i][2];
                }
                break;
        case 1: for (i=0;i<16;i++) {
                    rgbArray[i][0] = ciderpresscolors[i][0];
                    rgbArray[i][1] = ciderpresscolors[i][1];
                    rgbArray[i][2] = ciderpresscolors[i][2];
                }
                break;
        default: for (i=0;i<16;i++) {
                    rgbArray[i][0] = kegs32colors[i][0];
                    rgbArray[i][1] = kegs32colors[i][1];
                    rgbArray[i][2] = kegs32colors[i][2];
                }
                break;
        }


        for (i=0;i<16;i++) {
            rgbAppleArray[i][0] = r = rgbArray[i][0] >> 4;
            rgbAppleArray[i][1] = g = rgbArray[i][1] >> 4;
            rgbAppleArray[i][2] = b = rgbArray[i][2] >> 4;

            j = i*2;
            shrpalette[j+1] = r;
            shrpalette[j] = (g << 4) | b;

        }
}
```

There’s quite a lot going-on in the code above and we didn’t comment it either. It’s not that we didn’t want to comment; it’s simply because we wouldn’t have been able to read the code later on, so we sacrificed some so-called efficiency for readability and avoided the issue altogether.

You can see that this code uses something called a palidx. That's an index into 1 of 4 built-in 16 color palettes that I made from the DHGR and LGR color values in the Kegs32 emulator, CiderPress file viewer, and two versions of AppleWin that I have laying about. However, in the past I've remapped everything to everything else using these handy palette arrays, including HGR colors and VGA, EGA, and CGA colors and so-forth. Using this tabular technique is a solid and readable programming method, so we use it when we can to avoid "hard to read" calculations, or when in this case it is a necessity to achieve some results without confusing ourselves.

Since we will need an Apple II SHR palette for our output when we use built-in palettes, we also build that palette in the SHR's $0RGB - 32-byte palette format in our palette output buffer and save it away for later:

```
uchar shrpalette[32];
```

Initializing a Palette from an Existing SHR File

When we read palette 0 from an existing SHR file we already have **shrpalette** and don't need to build it for later. Here's how that works:

```
/* read SHR palette 0 from $C1 $0000 SHR PIC File */
sshort GetSHRPalette(char *shrfile)
{
    FILE *fp;
    sshort i,j,status = INVALID;
    ulong filesize;
    uchar r,g,b;

    if((fp=fopen(shrfile,"rb"))==NULL) {
        printf("Error Opening %s!\n",shrfile);
        return status;
    }
    for (;;) {
        fseek(fp, 0L, SEEK_END);
        filesize = ftell(fp);
        if (filesize != (ulong)32768L) {
            printf("%s is not a raw SHR file!\n",shrfile);
            break;
        }
        /* seek past image data to first palette in SHR file */
        fseek(fp,32256L,SEEK_SET);
        /* read SHR palette into buffer */
        if (fread((char *)&shrpalette[0],1,32,fp) != 32) {
            printf("Error Reading %s!\n",shrfile);
            break;
        }
        /* expand SHR palette from 12 bit to 24 bit gun values */
        /* and read into rgbArray */
        for (i=0;i<16;i++) {
            j = i*2;
            rgbAppleArray[i][0] = (shrpalette[j+1] &0xf);
            rgbAppleArray[i][1] = (shrpalette[j] &0xf0) >> 4;
```

```
            rgbAppleArray[i][2] = (shrpalette[j] &0xf);

            rgbArray[i][0] = (shrpalette[j+1] &0xf) << 4;
            rgbArray[i][1] = (shrpalette[j] &0xf0);
            rgbArray[i][2] = (shrpalette[j] &0xf) << 4;
        }
        status = SUCCESS;
        break;
    }
    fclose(fp);
    return status;
}
```

I am assuming that you already know about SHR Screen Image files, and that the code above is relatively straight forward. Whether you do or whether you don't, the notable part of this code is how we stuff the **rgbArray** and **rgbAppleArray** from the first palette in the SHR "raw" PIC file. We use the SHR palette values as-is for **rgbAppleArray** and promote them to 24-bit values from 12-bit values for **rgbArray**.

So whether we get SHR 16-color palette values from a built-in palette or from an existing SHR image or some other way, we have succeeded in having up to 16-colors to match-to the 16.7 million colors in our BMP input file. When selecting an SHR file we need to be guaranteed that we have 16 active colors in the first palette to match to. Code to provide some extra colors is too complicated for right now.

For right now, enough has been said about color matching, since we've solved most of this "mystery" already.

Converting a 24-bit BMP to an SHR Image Fragment

At some point, we likely decided to provide several types of output from our converter. We usually do that, because it takes very little extra time to write such code. But we also want to keep things simple when we are prototyping, so fancy stuff like compressing files and supporting output formats that will see no use are a waste of time, and are "indefinetly deferred". And don't forget that some joker will always pop-up later with some idea about what he would have liked, but since some ideas aren't from jokers at all, we need to write our code so it can be re-used and expanded while keeping it initially simple at the beginning.

For this version of our converter, we want to provide 3 major output options from our converted input:

- Primary Output of Binary Image Fragments in our new format.
- Secondary Output of a standard $C1 $0000 Screen Image File.
- Optional Secondary Embedded Arrays to be Included in cc65 and other programs.

Embedded arrays are not always desired, so these can just go to stdout (the console) when this option is set. To make a text file, console output of arrays can be redirected to disk.

The only time we may reasonably sometimes need a "standard" $C1 $0000 SHR Screen Image file is when we have a full-screen image so we make this output automatic when the BMP input file is 320 x 200, and unavailable when the BMP is smaller.

We also limit our input sizes to a sensible range because scaling and editing takes place outside our converter. Windows Paint and other BMP editors can scale as well as edit, so for a simple converter, we don't need to worry about re-inventing that wheel.

And so now it all "boils-down" to converting a BMP to our output formats:

```
/* reads a 24 bit BMP file in the range from 2 x 1 to 320 x 200 */
/* writes a $C1 $CC65 image fragment */
sshort Convert()
{
    FILE *fp,*fp2,*fp3;
    sshort status = INVALID, screenimage=INVALID;
    ushort x,y,i,spritewidth,spriteheight,packet;
    uchar r,g,b,c,c1,c2,bmpscanline[960];
    ulong pos;

    if((fp=fopen(bmpfile,"rb"))==NULL) {
        printf("Error Opening %s for reading!\n",bmpfile);
        return status;
    }
    /* read the header stuff into the appropriate structures */
    fread((char *)&bfi.bfType,
                 sizeof(BITMAPFILEHEADER),1,fp);
    fread((char *)&bmi.biSize,
                 sizeof(BITMAPINFOHEADER),1,fp);
    if (bmi.biCompression==BI_RGB &&
        bfi.bfType[0] == 'B' && bfi.bfType[1] == 'M' &&
        bmi.biPlanes==1 && bmi.biBitCount == 24) {

        spritewidth = (ushort) bmi.biWidth;
        spriteheight = (ushort) bmi.biHeight;

        if (spritewidth > 1 && spritewidth < 321 &&
            spriteheight > 0 && spriteheight < 201) status = SUCCESS;

        if (spritewidth == 320 && spriteheight == 200)
            screenimage = SUCCESS;
    }
    if (status == INVALID) {
        fclose(fp);
        printf("%s is in the wrong format!\n",bmpfile);
        return status;
    }
    if((fp2=fopen(spritefile,"wb"))==NULL) {
        printf("Error Opening %s for writing!\n",spritefile);
        fclose(fp);
        return INVALID;
    }
```

```
    if (screenimage == SUCCESS) {
       if((fp3=fopen(picfile,"wb"))==NULL) {
            printf("Error Opening %s for writing!\n",picfile);
            fclose(fp2);
            fclose(fp);
            return INVALID;
        }
    }
    packet = spritewidth * 3;
    /* BMP scanlines are padded to a multiple of 4 bytes (DWORD) */
    while ((packet % 4) != 0) packet++;
    /* SHR sprites are in SHR pixel-pairs - multiples of two
       for fast plotting etc. */
    spritewidth = (spritewidth/2);
    /* write 2 byte header */
    if (writeheader == 1) {
        fputc((uchar)spritewidth,fp2);  /* width in bytes */
        fputc((uchar)spriteheight,fp2); /* height in pixels */
    }
    /* write header values to stdout */
    if (quietmode == 0) {
        printf("#define %sWIDTH  %d\n",fname,spritewidth);
        printf("#define %sHEIGHT %d\n",fname,spriteheight);
        printf("#define %sSIZE   %d\n\n",fname,
                spritewidth * spriteheight);

    }
    /* write 32 byte shr palette */
    if (writepalette == 1) {
        fwrite((char *)&shrpalette[0],1,32,fp2);
    }
    /* write shr palette array values to stdout */
    if (quietmode == 0) {
    printf(
"/* Embedded SHR Image Fragment created from %s */\n\n",bmpfile);
        printf("unsigned char %sColorTable[32] = {",fname);
        for (x = 0; x < 32; x++) {
            if (x == 0 || x == 16) {
                printf("\n%3d,",shrpalette[x]);
                continue;
            }
            if (x == 31) {
                printf("%3d};\n\n",shrpalette[x]);
                break;
            }
            printf("%3d,",shrpalette[x]);
        }
    }
    /* write background color */
    if (backgroundcolor > -1 && backgroundcolor < 16) {
        fputc((unsigned char)backgroundcolor,fp2);
        if (quietmode == 0)
            printf("unsigned char %sBackgroundColor = %d;\n\n",
                    fname,backgroundcolor);
    }
```

```
    if (quietmode == 0) {
        printf("unsigned char %sPixelData[] = {\n",fname);
    }
    /* read BMP from top scanline to bottom scanline */
    pos = (ulong) (spriteheight - 1);
    pos *= packet;
    pos += bfi.bfOffBits;

    for (y=0;y<spriteheight;y++,pos-=packet) {
        fseek(fp,pos,SEEK_SET);
        fread((char *)&bmpscanline[0],1,packet,fp);
        for (x = 0,i = 0; x < spritewidth; x++) {
    /* make 12-bit color pixel-pairs from 24-bit RGB triples in BMP */
            b = bmpscanline[i]; i++;
            g = bmpscanline[i]; i++;
            r = bmpscanline[i]; i++;
            c1 = GetDrawColor(r,g,b); /* get nearest color */
            b = bmpscanline[i]; i++;
            g = bmpscanline[i]; i++;
            r = bmpscanline[i]; i++;
            c2 = GetDrawColor(r,g,b); /* get nearest color */
            c = (c1 << 4) | c2;
            /* write pixel-pair value to SHR image fragment */
            fputc(c,fp2);
            if (screenimage == SUCCESS) fputc(c,fp3);
            /* write pixel-pair value to stdout */
            if (quietmode == 0) {
                if (y == 0 && x == 0) {
                    printf("%3d",c);
                }
                else {
                    printf(",");
                    if (x%16 == 0) printf("\n");
                    printf("%3d",c);
                }
            }

        }
    }
    if (quietmode == 0) {
        printf("};\n\n");
    }
    fclose(fp);
    fclose(fp2);
    if (screenimage == SUCCESS) {
        c = 0;
        for (y = 0;y < 256; y++) fputc(c,fp3);
        fwrite((char *)&shrpalette[0],1,32,fp3);
        for (y = 0;y < 480; y++) fputc(c,fp3);
        fclose(fp3);
        if (quietmode == 1)printf("%s created!\n",picfile);
    }
    return SUCCESS;
}
```

Well, that certainly was easy. It's a good thing we wrote all that other stuff first☺

Starting with the beginning of the **Convert()** function we are reasonably careful to open our files in the order that makes sense. Without a BMP we have no output, so that comes first. But without being able to open our binary output file(s) we can't continue so we just bail!

BMP's for the most part store their data upside-down from the bottom-up but today we are in the mood to read a BMP in the SHR linear screen order from the top-down so we seek each BMP scanline and write our output in linear order. Aside from matching our mood, it makes the 3 different outputs easier to write.

We haven't commented heavily… just enough that we haven't destroyed the readability of the **Convert()** function. If we stick to this style when we have more complicated converters to write, others besides ourselves may even be able to read and re-use the code.

You can put my theory to a test now, and take a farewell look our simple converter. We need to move-on to our **main() program** and see how it ties the options together with the input and output code that we have already gone through.

The main() Program and Tying it Together

```
char *usage[] = {
    "Usage:   \"b2sprite sprite.bmp -options\" ",
    "*or*     \"b2sprite sprite.bmp background.shr -options\"",
    "Options: -t use CiderPress file attribute preservation tags.",
    "         Tags are turned-off by default.",
    "         -p (-p0,-p1,-p2,-p3) use built-in palette.",
    "         Kegs32 DHGR color palette (-p0) is the default.",
    "         background.shr over-rides built-in palette.",
    "         -b (-b0 to -b15) background color.",
    "         -np do not write palette to output.",
    "         -nh do not wrtite header to output.",
    "         -q disable quietmode (can be redirected to file).",
    "         creates embedded sprite array.",
    "Output:  sprite.sprite - $C1 $CC65 SHR image fragment.",
    NULL};

void pusage(void)
{
    sshort i;
    for (i=0;usage[i] != NULL;i++) puts(usage[i]);
}
```

Before we do **main()**, when we write command line programs like B2Sprite we always write a usage of some kind... sometimes we start with a comment or two. Some programmers don't say much. They think that users learn through osmosis during trial and error, or that if anyone is deserving of their program they can darned-well read the source code. Although all of us can experience similar violent mood swings during the aging process which begins at birth and ends when your source code is no longer good

for anything, we realize the tendency to reduce everything to a one-liner of intuitive code is best left to smarter people than us. So we update the little usage banner as we program the command-line options, and occasionally glance at it when we get lost in the code. We know it's really there for us so there is no use being in denial about it☺

As far as smarter people, I have yet to find one, but continue to find lazy people without looking very hard, starting with my mirror☺.

```
int main(int argc, char **argv)
{
    sshort idx,jdx,palidx=0,tags=0;
    uchar ch, *wordptr;

    if (argc < 2) {
        pusage();
        return (1);
    }
    /* getopts */
    if (argc > 2) {
        for (idx = 2; idx < argc; idx++) {
            wordptr = (uchar *)&argv[idx][0];
            ch = wordptr[0];
            if (ch != '-') {
                if (GetSHRPalette(wordptr) == SUCCESS) palidx = -1;
                continue;
            }
            ch = toupper(wordptr[1]);
            switch(ch) {

                case 'B': jdx = atoi((char *)&wordptr[2]);
                          if (jdx > -1 && jdx < 16)
                          backgroundcolor = jdx;
                          break;
                case 'P': if (palidx == -1) break;
                          jdx = atoi((char *)&wordptr[2]);
                          if (jdx > -1 && jdx < 4) palidx = jdx;
                          break;
                case 'Q': quietmode = 0;
                          break;
                case 'T': tags = 1;
                          break;
                case 'N': ch = toupper(wordptr[2]);
                          if (ch == 'P') writepalette = 0;
                          else if (ch == 'H') writeheader = 0;
                          break;

            }
        }
    }

    jdx = 999;
    strcpy(fname, argv[1]);
    for (idx = 0; fname[idx] != (uchar)0; idx++) {
        if (fname[idx] == '.') {
            jdx = idx;
```

```
        }
    }
    if (jdx != 999) fname[jdx] = (uchar)0;

    sprintf(bmpfile,"%s.bmp",fname);

    if (tags == 1) {
        sprintf(spritefile,"%s.SPRITE#C1CC65",fname);
        sprintf(picfile,"%s.SHR#C10000",fname);
    }
    else {
        sprintf(spritefile,"%s.SPRITE",fname);
        sprintf(picfile,"%s.SHR",fname);
    }

    if (palidx != -1)GetBuiltinPalette(palidx);
    InitDoubleArrays();
    if (Convert() == INVALID) return (1);

    if (quietmode == 1)printf("%s created!\n",spritefile);

    return SUCCESS;
}
```

Just for fun, we can read through **main()** and look for comments or we can use our time more wisely and review some of our options to see if our requirements for our little converter were addressed by our code. Other than that, we need to get busy to test our converter, and the best way to do that is get some cc65 demos together.

Command Line Options

Selecting Output Format Options

The mode320 SHR image fragment files produced by b2sprite will always contain a chunk of mode320 unpacked Pixel data, but the header, palette, and background color information is optional. Pixel data size can range from 2 pixels x 1 raster to 320 pixels x 200 rasters (an entire mode320 screen can even be stored as a fragment).

With all output options turned off, a fragment's file size is only 35 bytes larger than with all output options turned on, and only 34 bytes larger than the default. Meh!

Defaults

By default, a fragment has a header of byte-width x raster-height, followed by an SHR palette, followed by unpacked Pixel Data. A background color is not included by default.

Flexible Output Format

A variety of reasons exist for a programmer to use image fragments so their format is flexible. If a graphics program contains many image fragments in a fixed palette, the palette can be loaded in a background image, so the files themselves may not need a

palette. The width and height of image fragments may already be known to the program so may not need to be stored in the image fragment itself, and a header may not be needed. If the image fragment is to be rendered over a background, a "transparent" background color may be needed.

Background Color Option "-b"

Option "-b" enables writing a background color to an SHR image fragment output file.

Option –b by itself will make the first color in the palette the background color. Normally option –b is followed by a color number in the range of 0-15 (-b0, -b1, -b2, etc.).

If the image fragment is to be rendered over a background, a "transparent" background color may be needed so only 15 colors can be used to draw the image. If not, all 16 palette colors can be used.

No Header Option "-nh"

Option –nh disables writing width and height to the beginning of the SHR image fragment output file.

No Palette Option "-np"

Option –np disables writing a 32 byte palette to the SHR image fragment output file.

Selecting Input Options

There are two ways to set-up a palette for conversion of BMP colors to SHR colors;

- Use an existing palette from an SHR mode320 $C1 $0000 "PIC" file.
- Use one of 4 built-in palettes

Palette Option "-p"

This option selects 1 of 4 built-in SHR palettes. If a second file SHR PIC file of Type $C1 Auxiliary Type $0000 follows the BMP input file name on the command line, option "-p" will be cancelled and palette 0 from the "PIC" file will be used instead.

If option "-p" is not used at all, the kegs32 DHGR colors will be used for color-matching unless a "PIC" file palette 0 is used instead..

Variations of Option "-p"

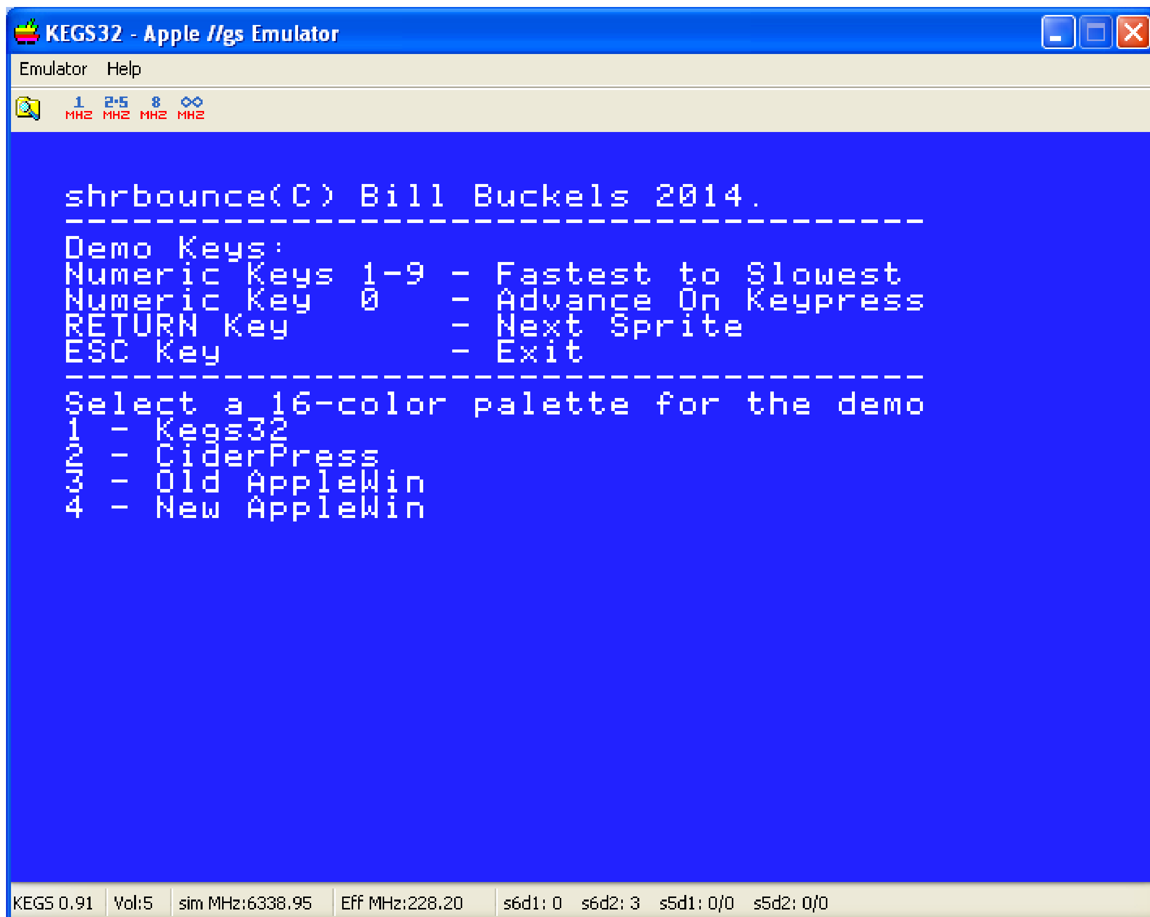
- -p0 Kegs32 DHGR Colors
- -p1 CiderPress DHGR Colors
- -p2 and –p3 Applewin LGR Colors

“PIC” File Palette Over-Ride Cancels Option “-p”

As noted, when a “raw” SHR $C1 $0000 Screen Image “PIC” file is named on the command-line following the input BMP file, palette 0 from the “PIC” file will be used for output, and option “-p” if any will be cancelled.

The preceding information in this document gives enough additional information to use our simple converter. So let’s get to work and do something with these new files!

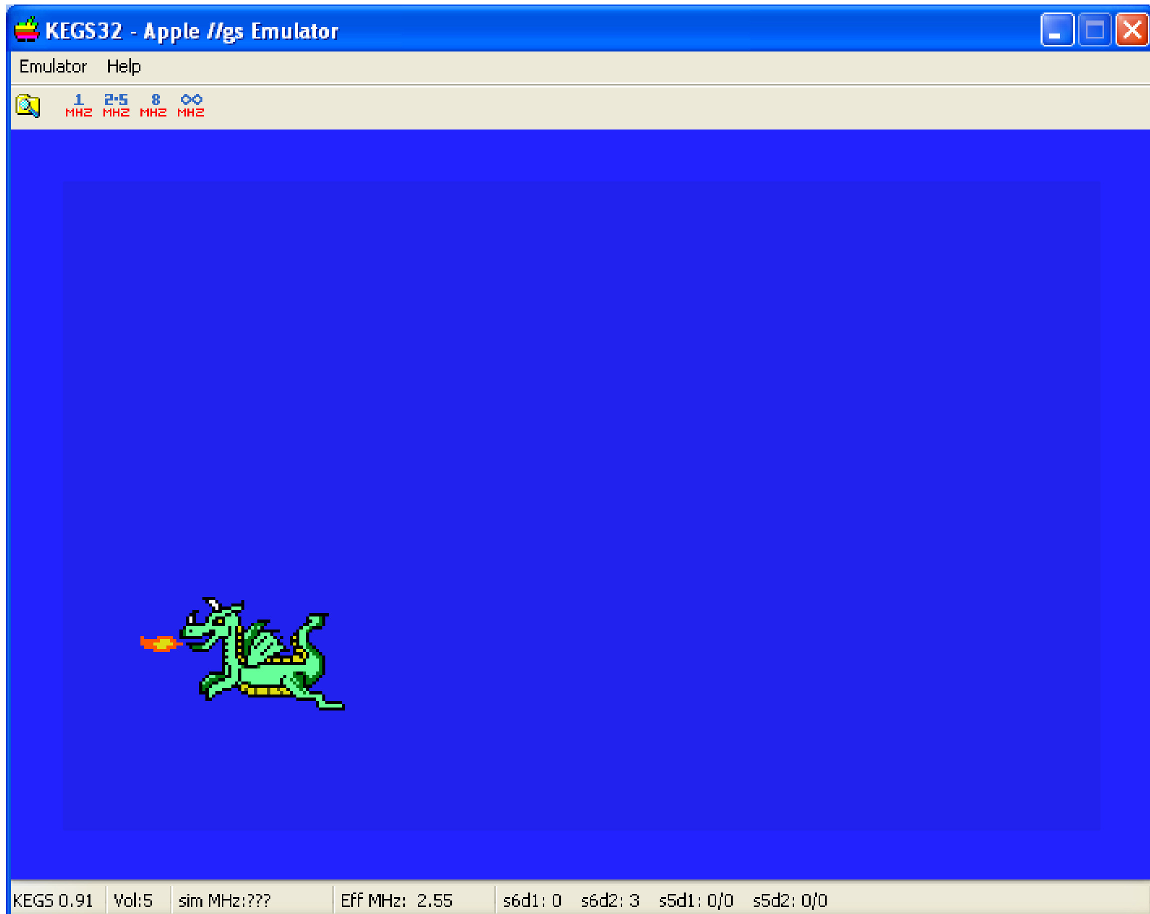
The Bounce Demo – Embedding an Image Fragment



What we have here is a cc65 Apple II SHR demo of the embedded output created using B2Sprite’s –q output option. The sprites in this demo could have been anything, but I lifted them from an old MS-DOS version of Broderbund Playroom, and edited the captured screens in Windows Paint to make something that could be bounced around.

There’s not much to this demo; you can see by the opening screen above that there are demo keys and you can also see that the 4 palettes that were used in our B2Sprite converter are used. So let’s start with the bouncing routine itself and see how it does what it does.

Bouncing a Sprite




```
/* bounce demo */
unsigned char FragDemo(unsigned char *rPixelData,
                       unsigned char *lPixelData,
                       unsigned width, unsigned height)
{

    int i,j, dx = 2, dy = 1;
    unsigned x1 = 0, x2 = (160 - width)*2;
    unsigned y1 = 0, y2 = (200 - height);
    unsigned char ch = (unsigned char)(demospeed + 48);
    unsigned char clearcolor = (rPixelData[0] >> 4);

    clear320(clearcolor);

 do {

    if (dx > 0)
PutFragment((unsigned char *)&rPixelData[0],width,height,x1,y1);
    else
PutFragment((unsigned char *)&lPixelData[0],width,height,x1,y1);
```

```
    if (demospeed == 0) {
        while (kbhit() == 0);
    }
    else {
        for (i = 0; i < (demospeed-1); i++)
            for (j=0;j<384;j++);
    }

    if ( y1 == y2) dy = -1;
    else if (y1 <1) dy = 1;
    if ( x1 == x2) dx = -2;
    else if (x1 < 1) dx = 2;

    x1 += dx;
    y1 += dy;

    if (kbhit() != 0) {
        ch = cgetc();
        if (ch == 13 || ch == 27) break;
        if (ch > 47 && ch < 58) demospeed = (ch - 48);
        while (kbhit() > 0)cgetc();
    }
 } while (ch != 27 || ch != 13);

 while (kbhit() > 0)cgetc();
 return ch;
}
```

The above code is a simple bounce algorithm that hits the SHR screen bounds of an x or y coordinate with an image fragment and then bounces to the reverse bound of the x or y co-ordinate, and changes direction, ad nauseum, until either ESC is pressed to end the program, or RETURN is pressed to rotate to the next sprite, and then the function repeats.

If you go back to the opening screen, you'll see that the numeric keys control the speed of the bounce, and pressing 0 will cause the bounce to proceed and stop on key press and key release. Any other key is ignored unless key press advance is active.

You can also see that when horizontal bounce direction changes, if the bounce is headed left, the left image fragment version is used and if the bounce is headed right, the right image fragment is used. The fragments used in this demo have a border around them to blot the previous placement of the fragment. This means that the background does not need to be saved and restored, so the bounce can just keep placing the fragment on the SHR screen non-stop as fast as possible.

The clearcolor is taken from the border color as well, so the screen background is the same color as the image fragment background.

The main control loop for **FragDemo()** is in the **main() program**.

Putting a Sprite on the SHR Screen

FragDemo() calls the **PutFragment()** to put a fragment on the SHR screen.

```
/* simple putimage function for this demo */
/* does not preserve the background */
#pragma optimize (push,off)
void PutFragment(unsigned char *PixelData,
                 unsigned width, unsigned height,
                 unsigned x, unsigned y)
{
    unsigned *src  = (unsigned *)0x3c;
    unsigned *dest = (unsigned *)0x42;
    unsigned y2 = y + height, offset = 0;
    while (y < y2) {
        src[0] = (unsigned) &PixelData[offset];
        offset+= width;
        src[1] = (unsigned) &PixelData[offset-1];
        dest[0] = (unsigned) (0x2000 + (y * 160) + (x/2));
        asm("sec");
        asm("jsr $c311");
        y++;
    }
}
#pragma optimize (pop)
```

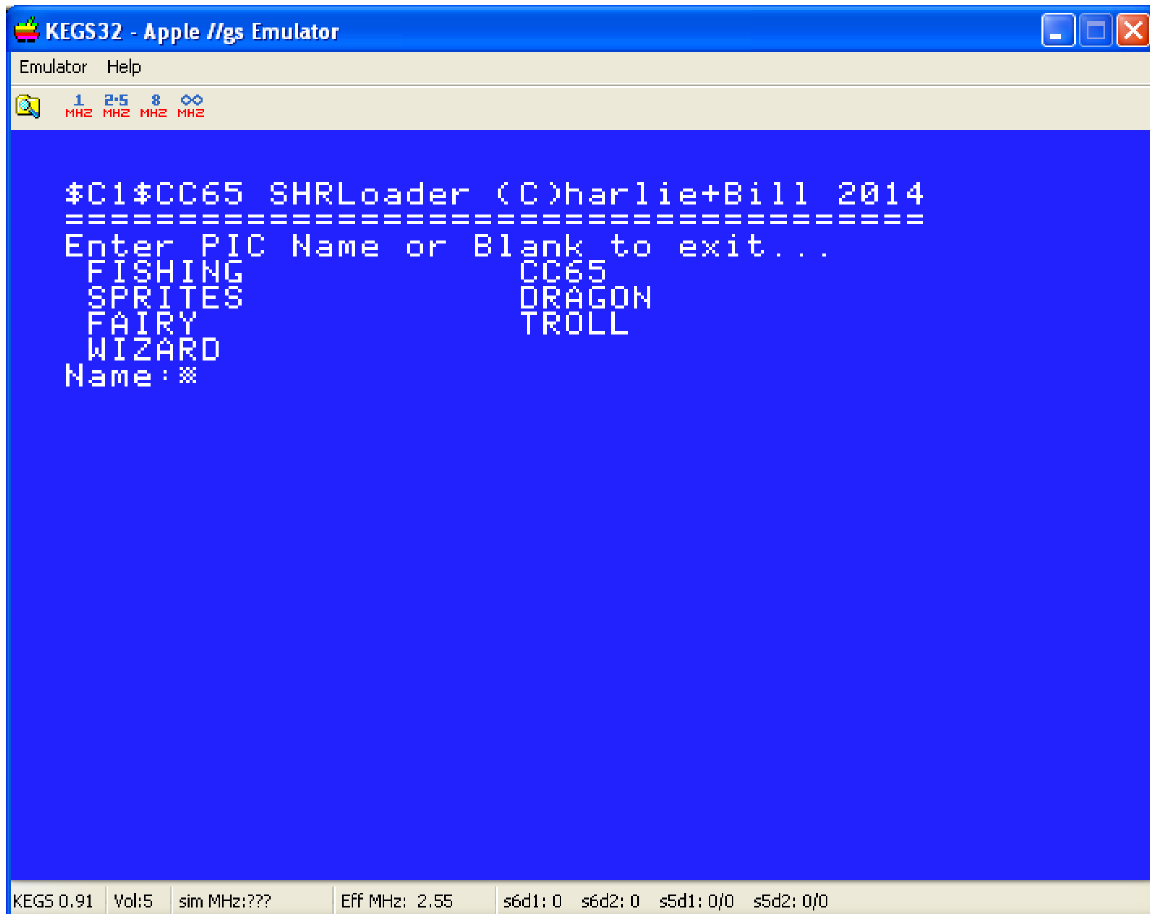
The Bounce Demo main() Program

```
/* speed 0 - advance on keypress */
/* speeds 1-9 fastest to slowest */
int demospeed = 3;
/* all the core routines etc. are included in the local headers */
#include "bounce.h"

int main(void)
{
    int i;
    unsigned char *ptr, ch;

    /* initialize text mode. stay in 40 column mode. */
    texton();
    clrscr();
    /* initialize empty palette and scb's */
    initbuffers();
    puts("shrbounce(C) Bill Buckels 2014.");
    puts("---------------------------------------");
    puts("Demo Keys:");
    puts("Numeric Keys 1-9 - Fastest to Slowest");
    puts("Numeric Key  0   - Advance On Keypress");
    puts("RETURN Key       - Next Sprite");
    puts("ESC Key          - Exit");
    puts("---------------------------------------");
    puts("Select a 16-color palette for the demo");
    puts("1 - Kegs32");
```

```
    puts("2 - CiderPress");
    puts("3 - Old AppleWin");
    puts("4 - New AppleWin");
    ch = cgetc();
    while (kbhit() > 0)cgetc();
    clrscr();
    /* turn shr on */
    shgron();
    /* now that the shr display is in auxiliary memory */
    /* clear the palette and point all the scbs to the
       first palette */
    clearpalette();clearscbs();clear320(0);
    /* set the palette that the user has selected */
    switch(ch) {
        case '1': ptr = (unsigned char *)&rgbkegs32[0];break;
        case '2': ptr = (unsigned char *)&rgbciderpress[0];break;
        case '3': ptr = (unsigned char *)&rgbawinold[0];break;
        default:  ptr = (unsigned char *)&rgbawinnew[0];break;
    }
    setpalette(ptr,0);
    i = 0;
    for (;;) {
        switch(i) {
          case 0:
          ch = FragDemo((unsigned char *)&rdragonPixelData[0],
                        (unsigned char *)&ldragonPixelData[0],
                        dragonWIDTH,dragonHEIGHT);
             break;
          case 1:
          ch = FragDemo((unsigned char *)&rfairyPixelData[0],
                        (unsigned char *)&lfairyPixelData[0],
                        fairyWIDTH,fairyHEIGHT);
             break;
          case 2:
          ch = FragDemo((unsigned char *)&rwizardPixelData[0],
                        (unsigned char *)&lwizardPixelData[0],
                        wizardWIDTH,wizardHEIGHT);
             break;
          case 3:
          ch = FragDemo((unsigned char *)&rtrollPixelData[0],
                        (unsigned char *)&ltrollPixelData[0],
                        trollWIDTH,trollHEIGHT);
             break;
         }
         if (ch == 27) break;
         i++;
         if (i > 3)i=0;
    }
    /* clear the palette and scbs then
       turn shroff and re-initialize text mode */
    clearpalette();
    clearscbs();
    shgroff();
    texton();
    clrscr();/* and exit */
    return 0;
}
```

Let's review what is happening in the bounce demo's **main() program**. We select a built-in palette, then call FragDemo() in a loop with a different embedded sprite on each iteration until ESC is pressed. As you can see by the code above, most of what is happening here is straight-forward. Let's look at our other demo now.

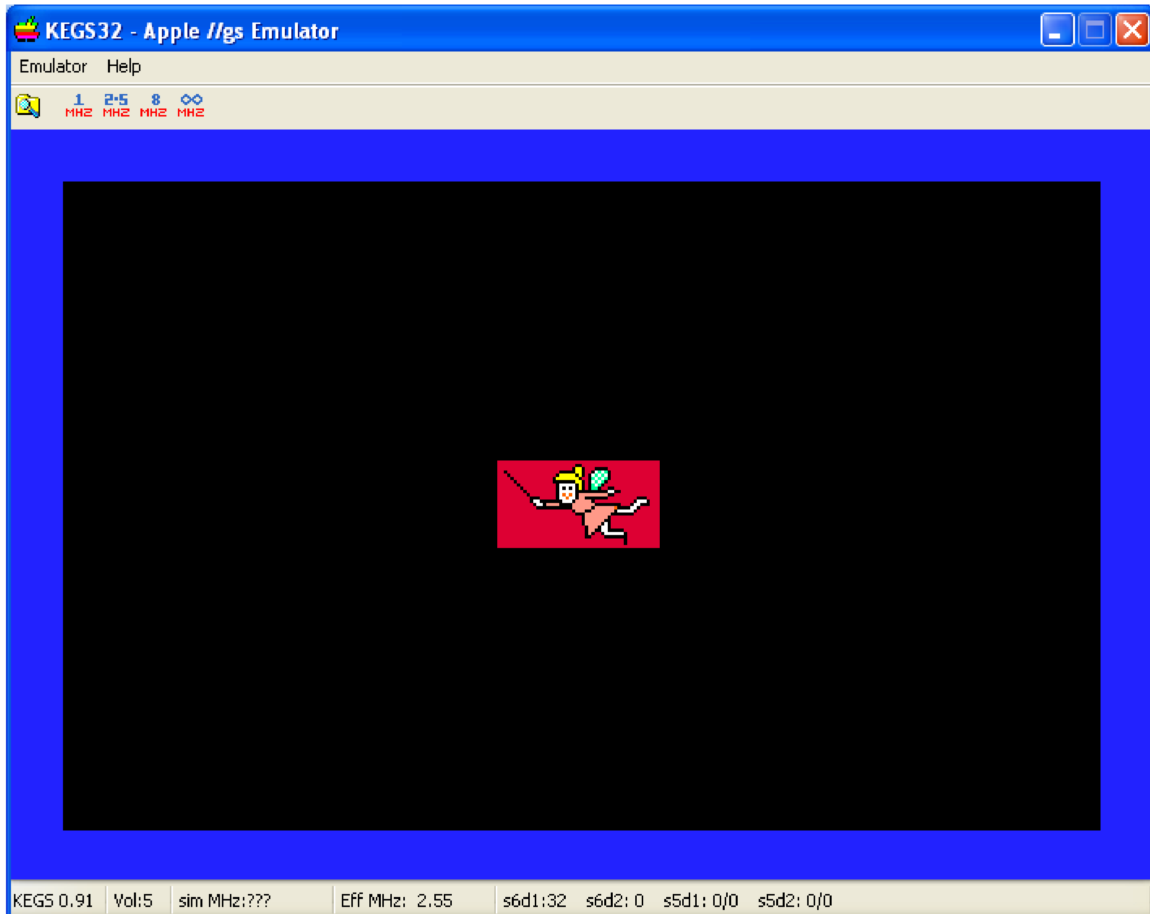
The Fraglode Demo - Loading an Image Fragment



What we have here is a cc65 Apple II SHR demo of loading the $C1 $CC65 output created using B2Sprite's default output option. The image fragments in this demo could have been of anything but I used a converter written by Jonas Grönhagen (STYNX) using the ImageMagick API to create some pairs of 320 x 200 BMP and SHR files with only 16 colors for some background images for the "banners" and threw-in the sprites from the bounce demo for "good luck". This was really more a test of our converter than of cc65 anyway. The loader itself is based on the piclode loader previously distributed for both Aztec C65 and cc65 so there wasn't much new to test on the cc65 side of the code.

There's not much to this demo; you can see by the opening screen above that the image fragments in the current directory have been listed and when we type-in one of these it will be displayed on the SHR display. So let's start with the display routine itself and see how it does what it does.

Displaying an Image Fragment File




```
/* load a raw SHR screen file */
int fraglode(char *name)
{

    FILE *fp;
    int i, c, status = -2;
    unsigned src1, src2, dest, width, height;

    fp = fopen(name,"rb");
    if (fp == NULL) return -1;
    /* clear the palette and scbs prior to loading */
    clearpalette();
    clearscbs();
    /* assume it's got a header */
    width = (unsigned)fgetc(fp);
    height = (unsigned) fgetc(fp);
    /* if not in range just bail */
    if (width > 160 || height > 200) {
        fclose(fp);
        return -3;
    }
```

```
    /* assume it's got a palette */
    c = fread((char *)&shrpal[0],1,32,fp);
    if (c != 32) {
        fclose(fp); return -4;
    }
    /* assume no background color - not supported for this loader */
    /* read image data */
    /* show full screen image after load */
    if (width == 160 && height == 200) {
        for (;;) {
            /* read image data */
            dest = 0x2000;  /* set destination to scanline 0 */
            for (i=0;i<4;i++) {
               c = fread((char *)0x2000,1,8000,fp);
               if (c != 8000)break;
               maintoaux(0x2000,0x3f3f,dest);
               dest += 8000;
            }
            if (c!=8000) break;
            status = 0;
            /* scb's already point to palette 1 */
            /* set palette 1 and show image */
            src1 = (unsigned) &shrpal[0]; src2 = src1 + 31;
            maintoaux(src1,src2,(unsigned)0x9e00+32);
            break;
        }

        fclose(fp);
        return status;
    }
    /* show image fragment during load */
    /* scb's all point to blank palette 1 */
    /* point only the scb's we need to display the fragment
       to palette 0 - leave the other lines alone */
    /* set palette 0 */
    src1 = (unsigned) &shrpal[0]; src2 = src1 + 31;
    maintoaux(src1,src2,0x9e00);
    /* clear the entire line buffer to black */
    memset(&scanline[0],0,160);
    for (;;) {
        /* center fragment */
        src1 = (200 - height)/2; src2 = src1 + height;
        dest =  (160 - width)/2;
        for (i=src1;i<src2;i++) {
           c = fread((char *)&scanline[dest],1,width,fp);
           if (c != width)break;
           putline(i);   /* put line */
           /* point scb for this line to palette 0 */
           setscb(i,0);  /* show line */
        }
        if (c!=width) break; status = 0; break;
    }
    fclose(fp);
    return status;

}
```

The above code is a simple loader program for image fragments that uses 2 different display methods.

- Full Screen Method – uses Palette 1
- Part Screen Method – uses Palette 0

In the code above, since the scanline control bytes (scb's) for the entire screen point to palette 1 and palette 1 is initially blank, the SHR screen is initially black.

Full Screen Method - You can see that when a full-screen SHR file is loaded, palette 1 is blank, until the entire file is loaded. Then palette 1 is set which displays the image. For efficiency, the file is read in blocks of 8000 bytes in 4 passes.

Part Screen Method – You can see that when an SHR image fragment is loaded palette 1 is also blank, and we set palette 0. Since the scanline control bytes (scb's) for the entire screen point to palette 1 the SHR screen is black. We display the scanlines in the file to the SHR screen one line at a time by pointing each scanline's scb to palette 0 as we read the file. Efficiency is gained for a small fragment since we don't need to clear the entire display area of the screen that is not used by the fragment.

Making an SHR File List

```
/* display up to 40 files in current directory */
int showpiclist(unsigned char d_type, unsigned d_auxtype)
{

    int cnt = 0;
    DIR *dir;
    struct dirent *dp;

    /* read files in current directory */
    if ((dir = opendir (".")) == NULL) return 0;

    while ((dp = readdir (dir)) != NULL) {
        if (cnt > 39) break;
        if (dp->d_type != d_type  ||
            dp->d_auxtype != d_auxtype) continue;
        printf(" %-18s", dp->d_name);
        cnt++;
        if(cnt%2 == 0) puts("");
    }
    if(cnt%1 == 0) puts("");
    closedir(dir);
    return cnt;
}
```

As we read through the ProDOS directory file, cc65 provides us with extended information about files in a directory using an extended POSIX-like File Entry in the form of the dirent structure which contains ProDOS specific information like File Type and Auxiliary Type which allows us to filter only the files we wish to display.

The Fraglode Demo main() Program

```
int main(void)
{
    char buffer[66];
    int c;

    /* initialize text mode. stay in 40 column mode. */
    texton();
    for (;;) {
        clrscr();
        puts("$C1$CC65 SHRLoader (C)harlie+Bill 2014");
        puts("======================================");
        puts("Enter PIC Name or Blank to exit...");
        showpiclist(0xc1,0xcc65);
        printf("Name:");
        gets(buffer);
        if (buffer[0] == 0)break;
        /* turn shr on */
        shgron();
        /* load the image */
        c = fraglode(buffer);
        /* if nothing went wrong then wait for a keypress */
        if (c == 0) cgetc();
        /* clear the palette and scbs then
           turn shroff and re-initialize text mode */
        clearpalette();
        clearscbs();
        shgroff();
        texton();
        /* if an error occurred, display the error and wait
           for a keypress. otherwise skip this part */
        if (c != 0) {
            clrscr();
            switch(c) {
                case -1: printf("Can't open %s\n",buffer); break;
                case -2: puts("Error reading image data!");break;
                case -3: puts("Error reading header!");break;
                case -4: puts("Error reading palette!");break;
                default: printf("fraglode returned %d\n",c);

            }
            puts("Press a key...");
            cgetc();
        }
    }
    return 0;
}
```

As you can see by the code above, most of what is happening here is straight-forward. A more challenging project might have been a sprite animation program over a background image or a scrolling background of image fragments with navigational sprites avoiding collisions with falling sprites, or sprite cars driving on a fragmented background track.

Cc65 SHR Core Routines

Both of our demo programs use essentially the same SHR core routines with some variations.

Auxiliary Memory Routines

AUXMOVE is generally a handy routine for any Apple IIe 6502 program (including a cc65 program like shrworld) that stores and retrieves data in auxiliary memory. AUXMOVE must be called with 80Store off. The carry flag determines the direction of the memory move:

```
/* move a block of data from main to auxiliary memory */
#pragma optimize (push,off)
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");

}
#pragma optimize (pop)
```

When the SHR display is active, AUXMOVE is not only handy, but a necessity, and the only way to move data between the SHR display in Auxiliary Memory and program memory in a cc65 8-bit 6502 C program without writing 65816 subroutines. When SHR is turned-on (see the **shgron()** function below), Auxiliary Memory is shadowed into the real screen memory by clearing bit 3 of the IIgs shadow register at $C035.

Setting-up the SHR Display

The following functions are used to set-up the SHR display:

SHR Soft Switches

```
#define  gswitch ((unsigned char*)0xC029)
#define  shadow  ((unsigned char*)0xC035)
/* graphics - save previous setting of gswitch */
unsigned char gsave = 0xff;
/* shadow memory - save previous setting of shadow switch */
unsigned char ssave = 0xff;
```

```
/* turn-on SHR */
void shgron(void)
{
    gsave = gswitch[0];      /* save previous gswitch settings */
    gswitch[0]= gsave | 0xc0;/* shgr on  - set bits 6 and 7 */
    ssave = shadow[0];
    /* Bank $01 is shadowed into $E1 by clearing bit
       3 of the Shadow register at $C035 */
    shadow[0] = ssave & 0xf7;

}

/* turn-off SHR */
void shgroff()
{
    gswitch[0]=gsave; /* shgr off - restore previous setting */
    shadow[0] =ssave;
}
```

SHR Initialization

There is more to setting-up the SHR display than just setting some “soft-switches”. Since SHR Video Memory is divided into 3 parts in Auxiliary Memory starting at $2000, with the scanline control bytes (scb’s) and the palettes following 32000 bytes of image data, all three of these areas need to be initialized before using the SHR display.

SHR Initialization for Bounce Demo

```
/* buffers for 16 palettes and 200 scanline control bytes */
unsigned char palbuf[512], scbbuf[256];

void clearpalette()
{
    unsigned src1 = (unsigned)&palbuf[0];
    unsigned src2 = src1 + 511;
    /* zero palette in auxiliary memory */
    maintoaux(src1,src2,0x9e00);
}

void clearscbs()
{
    unsigned src1 = (unsigned)&scbbuf[0];
    unsigned src2 = src1 + 199;
    /* zero scanline control bytes in auxiliary memory */
    maintoaux(src1,src2,0x9d00);
}

void initbuffers(void)
{
    memset((char *)&palbuf[0],0,512);
    memset((char *)&scbbuf[0],0,256);
}
```

As you can see in the code above, the palette and scanline control bytes are buffered in program memory and are initially cleared, then moved to the SHR display in Auxiliary Memory. Doing so has the same effect as clearing the SHR display to black, because effectively an all-black palette has been created, and all 200 lines of the scanline control bytes have also been cleared. So they initially point all 200 scanlines to palette “zero”; the first of 16 “blank” palettes in the SHR palette memory.

With the palettes cleared, we don’t see what was in the SHR display when we started. And we can put whatever we want in the SHR’s image data memory area without it being displayed, until we finally set-up the scanline control bytes to point our scanlines to specific palettes other than the first palette if we need to, and then finally when we put the color values into our palette(s), whatever is in the image data area will be displayed.

SHR Initialization for Fraglode Demo

The code below works on the same principle as the Bounce Demo’s initialization code shown above, but our image fragment loader has some special needs and desires; for one thing the loader never clears the shr screen (but the bounce demo needs to).

Instead of clearing the screen, the loader clears the palette between loads to clear the screen, and then points every scanline in the SHR’s Pixel data area to the cleared palette by setting the 200 scb’s to 1 between loads. Our loader doesn’t care what’s in the screen (but the bounce demo wants color in the screen).

Our loader doesn’t need to clear the screen when it loads a screen size image fragment file as previously shown in the **fraglode()** function; all it needs to do is quickly load the SHR image with a blank palette then load the 16 color palette that was in the file into palette 1 in SHR memory. The scb’s are already pointing to palette 1.

For an image fragment smaller than the screen, as previously noted, a different method is used. The scb’s start-off pointing to palette 1 and the 16 color palette is read from the fragment file into palette 0. Rather than clear the entire screen, a scanline buffer is initially cleared in the loader, then the fragment is read from file line by line into the portion of the cleared buffer that is relatively justified to horizontal mid-screen. Each time a line is read, it is displayed on the SHR screen using the **putline()** function shown below, and only the scb for that line is changed to point to palette 0 instead of palette 1 using the **setscb()** function shown below to turn-on the line. For a small fragment this is much quicker than clearing the whole screen to display a few scanlines.

```
unsigned char palbuf[512], shrpal[32], scbbuf[256], scanline[160];

void clearpalette() {
    unsigned src1 = (unsigned)&palbuf[0];
    unsigned src2 = src1 + 511;
    /* zero palette in auxiliary memory */
    memset((char *)&palbuf[0],0,512);
    maintoaux(src1,src2,0x9e00);
}

void clearscbs()
{
    unsigned src1 = (unsigned)&scbbuf[0];
    unsigned src2 = src1 + 199;

    /* zero scanline control bytes in auxiliary memory */
    memset((char *)&scbbuf[0],1,200);
    memset((char *)&scbbuf[200],0,56);
    maintoaux(src1,src2,0x9d00);

}

#pragma optimize (push,off)
void setscb(unsigned y, unsigned palidx)
{

    unsigned *src  = (unsigned *)0x3c;
    unsigned *dest = (unsigned *)0x42;

    scbbuf[y] = (unsigned char)palidx;
    src[0] = src[1] = (unsigned)&scbbuf[y];
    dest[0] = (unsigned)0x9d00 + y;
    asm("sec");
    asm("jsr $c311");

}
#pragma optimize (pop)

#pragma optimize (push,off)
void putline(unsigned y)
{

    unsigned *src  = (unsigned *)0x3c;
    unsigned *dest = (unsigned *)0x42;

    src[0] = (unsigned) &scanline[0];
    src[1] = (unsigned) src[0]+159;
    dest[0] = (unsigned) (0x2000 + (y * 160));
    asm("sec");
    asm("jsr $c311");

}
#pragma optimize (pop)
```

Clearing the SHR Screen for Bounce Demo

Clearing SHR's image data area (the SHR screen) is done by transferring a buffer of "colored" bytes of "pixel pairs" from main memory to the SHR display in auxiliary memory.

```
void clear320(unsigned char color)
{
unsigned char paircolor = (unsigned char) (color << 4 | color);
unsigned y;
unsigned src1 = (unsigned) &scanline[0], src2 = src1+159;

    /* put pixel pair array on SHR display */
    /* fill scanline buffer with color pairs */
    memset((char *)&scanline[0],paircolor,160);
    /* wipe-down 200 scan lines with color */
    for (y=0;y<200;y++) {
        maintoaux(src1,src2,(unsigned) (0x2000 + (y * 160)));

    }
}
```

SHR Palettes for Bounce Demo

In the `clear320()` code above, you saw the creation of a colored "pixel pair":

```
paircolor = (unsigned char) (color << 4 | color);
```

Selecting 16 Colors

The bounce demo provides a choice of four 16 color palettes when the program starts. These palettes are based on Windows programs widely used by today's Apple II users.

```
/* 4 - palette options */
/* these are the rgb values of the lo-res colors from the AppleWin and
Kegs32 Emulators and from the CiderPress File Viewer. two sets of
AppleWin Colors are provided: one from an older version and one from a
newer version. the routines in this demo are based on a fixed palette
so it was convenient to use known colors and color order.*/

unsigned char rgbawinold[48] = {…};
unsigned char rgbawinnew[48] = {…};
unsigned char rgbkegs32[48] = {…};
unsigned char rgbciderpress[48] = {…};
```

One of the reasons that I stayed with LORES colors and color order is to be consistent; I have used these in my other cc65 graphics programs and documentation.

Setting an SHR Palette

```
/* sets a 12 bit color palette line from an array of 24 bit color
values */
/* rgbpal is a 48 byte character array of 16 - r,g,b values */
/* palidx is the SHR palette number in the range of 0-15 */
void setpalette(unsigned char *rgbpal,int palidx)
{
    unsigned char r,g,b;
    int i,j,k;
    unsigned src1 = (unsigned)&shrpal[0], src2 = src1 + 31;

    /* build 12 bit palette line of $0RGB color entries
       from 24 bit r,g,b values */
    for (i=0,j=0,k=0;i<16;i++,j+=3,k+=2) {
        r = rgbpal[j]   >> 4;
        g = rgbpal[j+1] >> 4;
        b = rgbpal[j+2] >> 4;
        shrpal[k] = (unsigned char)((g << 4) | b);
        shrpal[k+1] = r;
    }

    /* move palette line to palette */
    maintoaux(src1, src2,(unsigned)(0x9e00 + (palidx * 32)));

}
```

Building the cc65 Demo Programs

The Bounce demo and the Fraglode demo are cc65 binary programs with a starting address at $4000. They are launched using Oliver Schmidt's LOADER.SYSTEM ProDOS 8 SYS program. For consistency they come with the same linker configuration and build environment as my other cc65 SHR demo programs. The build environment is complete with a gcc compatible MAKEFILE. There is no need to document it here. For more info, review the MAKEFILE and the other baggage that comes with them.

These SHR demos are provided with monolithic source code with many of the routines in header files; they *DO NOT* link to any special library. They only use the libraries provided with the current cc65 snapshot. This is because all of this is still under development and will not be put into any library or tgi driver until the end of this project if at all. Since we are only at the start of this project, I will continue to use this monolithic format for distributing and improving everything and making it available as I go.

Download These Projects

B2Sprite Doc	**http://www.appleoldies.ca/cc65/docs/shr/b2sprite.pdf**
B2Sprite Utility	**http://www.appleoldies.ca/cc65/programs/shr/b2sprite.zip**
Bounce Demo	**http://www.appleoldies.ca/cc65/programs/shr/shrbounce.zip**
Fraglode Demo	**http://www.appleoldies.ca/cc65/programs/shr/fraglode.zip**

Additional Notes

What you have here is one small step for a compiler but a giant leap for a man. You now have the basis for potentially developing an SHR game of some kind in cc65 or something equally non-trivial.

You also have the basis for writing a graphics converter that does palette matching in a straight forward manner in Ansi C. This too is non-trivial. Congratulations!

You have my solemn oath that I have tried to cram as much useful information as possible into this document without using useless point form notation in "Broken English". It's pretty hard to beat some of the "Plain English" articles of the day like **Clearing Some Mist from Super HiRes** but you can't blame a guy for trying☺

How I Got Into This Mess

In May 2013, in the CSA2 Usenet group, "Charlie" announced the arrival of Super Hi-Res Graphics capability for the Carte Blanche card in Apple //e emulation mode (http://noboot.com/charlie/cb2e_p2.htm). According to "Charlie", "The SHR works pretty much as it does on an Apple IIgs, so you can write 8-bit programs that work in either."

That post really caught my interest. Up to that point most of my Apple II retro-computing focus had been mainly concerned with doing cross-platform development in Aztec C65 for the Apple //e. I was barely conscious that my Apple IIgs had better graphics capabilities than my Apple //e and my pre-occupation with the Aztec C65 compilers had led me backwards and sideways while my GS "sat on ice". My Carte-Blanche has also been "on ice" since it arrived. There just seems to be so little time.

"Aha!" thought I! "I can write an 8-bit SHR loader in Aztec C65 that runs on both the Apple //e and the Apple IIgs!" so I contacted Charlie and he set-me-up with an ever expanding reference library about all things SHR, and was very hands-on, helping me write those first loaders, testing and debugging with me, sending me "suggested" changes… until all things SHR were singing and dancing in Aztec C65.

By the time the winter of 2013 "rolled-around", I had become quite familiar with the history and details of SHR graphics. Antoine Vignau of Brutal Deluxe Software (http://www.brutaldeluxe.fr/products/apple2gs/convert3200.html) (and many others) had also contributed to my "crash course". And every time I thought I was done with SHR and extending video on the Apple II, someone else would post something in csa2 on a related note and it would get me started back on SHR again.

When Jonas Grönhagen (STYNX) did a series of posts about the Apple Video Overlay Card (VOC), and confirmed that my SHR loaders also worked on the VOC, combined with my other "discovery" of the Brooks mode3200 format which "Charlie" had seeded

right from the start along with Antoine and work by Andy McFadden http://ciderpress.sourceforge.net/ , I wrote my first SHR graphics converter (BMP2SHR), finishing the first version at the end of December 2013. After "getting it out there", I immediately started on the second version, getting even older and wiser, while I bathed in the topic of SHR and all the help I was getting from the csa2 folks. Through their help (combined with my own research), I managed to accumulate enough material on SHR to fill the rest of at least one mere-mortal lifetime, so after going through it all, it took me at least a month of coding, testing, and documenting in my spare time to get BMP2SHR version 2.0 together, with all its additional features, and bug fixes.

That saga continues to this very day, with several more SHR converters and the like "under my belt" now. But while I was busy knitting SHR programs, other tempests were also brewing in my retro-teapot!

The Aztec C65 cross-compilers and most of the utilities I have written for them run in MS-DOS. Over the past year or so as I became more of an expert on the Apple II, I also realized more and more the failings of Aztec C65 programs and Aztec C65 itself when compared to the cc65 cross-compiler. I also realized more and more that MS-DOS is as dead as the Apple II, and even emulators like DOSBox don't always work with the oldest Aztec C65 compilers, like the Commodore 64 compiler that I put together and distribute from the Aztec C Website:

http://www.aztecmuseum.ca

In 2009 I had also decided to port all of my Aztec C65 work to the cc65 cross-compiler. I wasn't quite finished my forensics, as the mysteries of the past continued to fill my sleepless nights with the morbidities that seemed to unravel within the bowels of that fascinating dissection. But by May 2014 I'd had quite enough of myself, and decided to do my utilities exclusively in the MinGW gcc compiler where possible, and abandon MS-DOS where possible. It was time to port everything else related to programming the Commodore 64 and Apple II to cc65.

MS-DOS support is quickly vanishing from the planet. Microsoft compilers produce bloatware so MinGW which is gcc compatible makes sense for two good reasons. Cc65 writes generally faster and smaller code than Aztec C65, so this move comes none too soon.

One small step for a compiler but a giant leap for a man.

Bill Buckels
bbuckels@mts.net
July 2014