Part II

Being One With The Pixel

Welcome to Part II of *Programming The Nintendo Game Boy Advance: The Unofficial Guide*. Part II includes three chapters that are dedicated entirely on programming the graphics system of the Game Boy Advance. These chapters cover the bitmap-based video modes, tile-based video modes (including coverage of backgrounds), and a chapter on sprites. Going into the first chapter of this part, you will learn to draw pixels on the screen in each video mode. Going out of the last chapter in this part, you will have learned how to scroll backgrounds and rotate, scale, and draw alpha-blended and animated sprites over backgrounds.

Chapter 5

Bitmap-Based Video Modes

This chapter is an introduction to basic graphics programming for Game Boy Advance, explaining the various graphics modes that are available (with the pros and cons of each) and showing how to draw pixels, lines, circles, bitmaps, and other shapes on the screen in each mode. The GBA has six different video modes that you may use, each with different resolutions, color depths, and capabilities. This chapter will teach you how to use the bitmap-based video modes 3, 4, and 5. The tile-based modes (0, 1, and 2) are covered in the next chapter. One of the goals of this chapter is to provide you with a collection of functions for rendering graphics in any of the three bitmapped video modes, which you may then copy and paste into other projects.

Here are the main topics of this chapter:

- Introduction to bitmapped graphics
- Working with mode 3
- Working with mode 4
- Working with mode 5
- Printing text on the screen

Introduction to Bitmapped Graphics

Understanding bitmapped graphics is the key to GBA game development, because when it comes down to it, all games are based on bitmap images of one format or another. The most important thing to consider when choosing a bitmap format is the amount of loss. A lossy format has a high compression ratio that keeps the file small, but at the cost of image quality. Examples include JPG and GIF. When it comes to games, quality is absolutely essential, so a nonlossy format must be used. Examples include BMP, PNG, PCX, and TGA. You may have a preference for one or another nonlossy format, but you will need a tool to convert an image file into a GBA image.

Now, by *image* what I am really referring to is a raw format that is actually compiled into your program. This differs greatly from what you may be used to, where in a standard operating system (Windows, Linux, Mac) you may store game graphics in one or more files and load them when needed. However, the GBA doesn't have a file system—everything must be stored inside the ROM! When I was first learning about GBA graphics programming, I thought perhaps the GBA used a proprietary bitmap file format stored in the ROM in a special resource of some kind. But that is not right; it's actually much simpler than that. The bitmaps used for backgrounds, tiles, and sprites in a GBA program must be converted to a C file as an array of numbers! Archaic, isn't it? Well, that's the console world for you. What this means, unfortunately, is that anytime you need to make a change to game graphics, you must run a utility program to convert the bitmap file to a source code file, such as monster.c.

In addition to a C byte array, graphics may be converted to raw binary format and linked directly into a game. For the sake of clarity, this book uses C arrays exclusively. In a practical sense, there is no real advantage one way or the other, performance-wise, although it is much easier to just include the C array for a bitmap in a project.

For example, here is the file from the ShowPicture program presented in Chapter 3:

```
//
// bg (38400 uncompressed bytes)
//
extern const unsigned char bg_Bitmap[38400] = {
0x4f, 0x3b, 0x23, 0x23, 0x2c, 0x1f, 0x22, 0x2d, 0x2f, 0x2f, 0x2f, 0x2f,
0x31, 0x2f, 0x2d, 0x2f, 0x23, 0x1d, 0x2d, 0x4c, 0x37, 0x38, 0x2f, 0x2d,
0x22, 0x1f, 0x2f, 0x2d, 0x2d, 0x1d, 0x22, 0x2d, 0x23, 0x23, 0x33, 0x2f,
0x2f, 0x2d, 0x34, 0x2f, 0x2f, 0x2d, 0x2c, 0x23, 0x23, 0x2d, 0x2d, 0x2d,
0x22, 0x1f, 0x2f, 0x2b, 0x23, 0x2d, 0x2e, 0x2d, 0x1d, 0x23, 0x38, 0x2b,
.
.
.
0xe1, 0xe4, 0xc7, 0xbf, 0xc3, 0xf2, 0xf2, 0xf6, 0xf6, 0xf7, 0xf8, 0xf9,
0xf8, 0xf8, 0xf8, 0xf8, 0xf9, 0xf9, 0xf8, 0xf8, 0xf8, 0xf8, 0xf2, 0xf8,
0xf8, 0xf8, 0xf2, 0xf6, 0xf6, 0xf2, 0xf2, 0xf2, 0xf2, 0xf6, 0xf2, 0xd9,
0xe1, 0xd6, 0xd9, 0xd9, 0xd9, 0xd9, 0xc9, 0xf6, 0xf6, 0xf2, 0xf6, 0xf8,
0xf8, 0xf8, 0xf8, 0xf8, 0xf8, 0xf8, 0xf8, 0xf9, 0xf9, 0xf9
};
```

I have listed only a portion of the actual file, which is quite large. This bitmap image was originally called bg.bmp (shown in Figure 5.1) and was converted by a program called gfx2gba.exe. This program specifically converts Windows .bmp files to GBA format. Actually, I shouldn't call it GBA format, because any C program could use it.

Figure 5.1
The bg.bmp bitmap image.

As you can see from the header, there are 38,400 bytes in this bitmap file. Regardless of the source image format, this is a raw format, just an array of 16-bit hexadecimal numbers, each of which represents a single pixel. If you do a little deductive mathematics, you can make an educated guess on the image size. A screen resolution of 240 x 160 = 38,400 pixels, so this must be a full-screen background image used in a mode 3 program. Of course, bitmaps are not limited to the screen resolution. Indeed, a bitmap can be quite huge, particularly when working with an animated sprite. For example, Figure 5.2 shows an animated explosion sprite, with a resolution of 524 x 142, which is much larger than the physical screen. However, when displayed on the screen, the explosion is 64 x 64. That's actually a rather large sprite for the GBA screen (approximately one-fourth of the screen), but it looks terrific when animated! I'll cover sprites in more detail in Chapter 7, "Rounding Up Sprites."

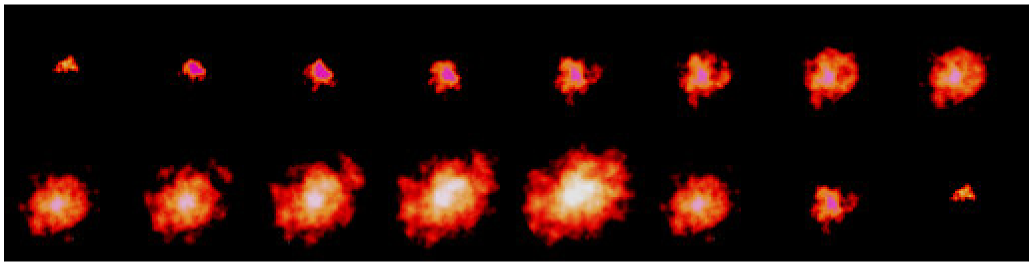


Figure 5.2

An animated sprite comprises multiple frames stored in a single bitmap file.

Selecting the Ideal Video Mode

When it comes to designing a new game, the resolution for your game is an important consideration early on, because the modes supported on the GBA differ greatly. Table 5.1 shows the three bitmap modes and their settings.

Table 5.1 Video Modes

Mode	Resolution	Color Depth	Double Buffered
3	240 x 160	15	No
4	240 x 160	8	Yes
5	160 x 128	15	Yes

Hardware Double Buffer

Modes 4 and 5 require less memory for the video buffer, so they are able to use two buffers for high-speed flicker-free screen updates. Mode 3, on the other hand, is both high in resolution and high in color depth, so it has enough video memory for just a single video buffer; thus double buffering is not available. However, there's no reason why you can't create your own double buffer in EWRAM and then perform a fast copy to the video buffer during the vertical retrace. A double buffer directly in video memory is the fastest method, of course, but using EWRAM is fast enough for most needs. If IWRAM were larger than 32 KB, it would be a great place to store the double buffer, but we need 75 KB for a mode 3 buffer. Since EWRAM has 256 KB available, it is the only viable option. Even if you are skilled enough to divide the second buffer between the 32 KB IWRAM and the remaining 20 KB or so leftover in video memory, that still isn't enough memory for the mode 3 buffer. I'll show you how to create a mode 3 double buffer later in this chapter.

Horizontal and Vertical Retrace

The screen refreshes at a fixed interval in the horizontal and vertical directions. What this means is that there is a short period after the LCD draws a horizontal line while it repositions to the next line and a somewhat longer (but still relatively short) blank period after the last pixel has been drawn while the video chip repositions back up to the upper-left corner of the screen. When dealing with a cathode ray tube (CRT) monitor for a PC, or perhaps a television, there are physical electron guns shooting electrons through the phosphorous layer of the screen, causing each pixel to light up for a brief period. Some of the older displays and TVs had a problem with ghosting because the phosphorous layer's pixels would remain lit too long, but modern displays don't usually have this problem any longer. The vertical retrace is really the only thing we're interested in for game programming, because this provides a window of opportunity. Any screen writes done during the vertical refresh are displayed as a whole, providing a nice consistent screen (and even frame rate).

Working with Mode 3

Video mode 3 is probably the most appealing mode, because it is the highest quality mode available on the GBA, with a resolution of 240 x 160 and a 15-bit color depth providing up to 32,767 colors on the screen at once. What this means is that your game's graphics will look extraordinary, with beautiful shades and intricate backgrounds. In contrast, mode 4 is limited to 256 colors, and while mode 5 also has 15-bit color, it is limited in resolution. The only problem with mode 3 is that you must provide your own double buffer. This isn't a problem, but the buffer does take away from available EWRAM that may be needed in the game (for instance, sound effects and music). However, let's just try a few things and see how it goes. This may be a factor in your decision to go with one video mode over another. For instance, if you are writing a high-speed 3D game that needs lots of memory, I would recommend mode 5 because it is much, much faster, due to the lower resolution, while still maintaining quality.

Drawing Basics

Let's dive right into the code for mode 3 and get something up on the screen. You have already seen an example of mode 3 in action from the DrawPixel and FillScreen programs in Chapter 4. However, one of those used Hamlib and the other dealt only with pixels. Now it's time for your formal training in pixel management.

Drawing Pixels

It's very easy to draw pixels in mode 3, because each pixel is represented by a single element of the video buffer (which comprises unsigned shorts). To draw a pixel, use an algorithm that has been passed down from one game programmer to another throughout history:

```
videoBuffer[y * SCREENWIDTH + x] = color;
```

Here's a complete reusable function for drawing pixels in mode 3:

```
void DrawPixel3(int x, int y, unsigned short color)
{
    videoBuffer[y * 240 + x] = c;
}
```

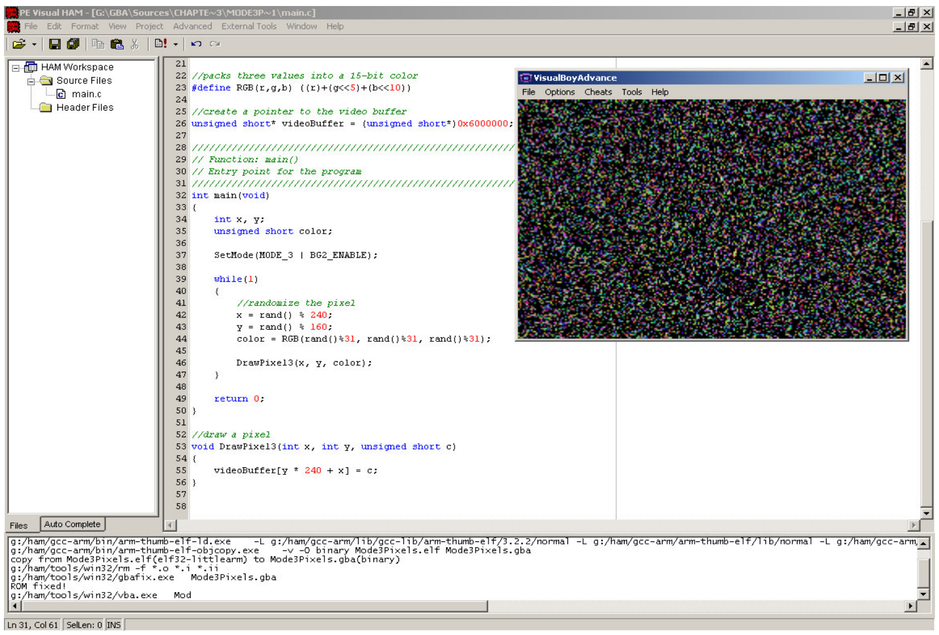
The Mode3Pixels program demonstrates how to draw random pixels on the mode 3 screen. Create a new project in Visual HAM called Mode3Pixels and replace the default code in main.c with the following code listing:

```
//////////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 5: Bitmap-Based Video Modes
// Mode3Pixels Project
// main.c source code file
//////////////////////////////////////////////////////////////

//add support for the rand function
#include <stdlib.h>

//declare the function prototype
```

```
void DrawPixel3(int, int, unsigned short);

//declare some defines for the video mode
#define REG_DISPCNT *(unsigned long*)0x4000000
#define MODE_3 0x3
#define BG2_ENABLE 0x400

//changes the video mode
#define SetMode(mode) REG_DISPCNT = (mode)

//packs three values into a 15-bit color
#define RGB(r,g,b) ((r)+(g<<5)+(b<<10))

//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;

/////////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
/////////////////////////////////////////////////////////////
int main(void)
{
    int x, y;
    unsigned short color;

    SetMode(MODE_3 | BG2_ENABLE);

    while(1)
    {
        //randomize the pixel
         x = rand() % 240;
        y = rand() % 160;
        color = RGB(rand()%31, rand()%31, rand()%31);
```

```
            DrawPixel3(x, y, color);
        }

        return 0;
    }

    //////////////////////////////////////////////////////////////////
    // Function: DrawPixel3
    // Draws a pixel in mode 3
    //////////////////////////////////////////////////////////////////
    void DrawPixel3(int x, int y, unsigned short color)
    {
         videoBuffer[y * 240 + x] = color;
    }
```

The output from the Mode3Pixels program is shown in Figure 5.3.


PE Visual HAM - [G:\GBA\Sources\CHAPTE~3\MODE3P~1\main.c]
File Edit Format View Project Advanced External Tools Window Help
HAM Workspace
Source Files
main.c
Header Files
Files
Auto Complete
VisualBoyAdvance
File Options Cheats Tools Help
Ln 31, Col 61
SelLen: 0
INS

```

//packs three values into a 15-bit color
#define RGB(r,g,b) ((r)+(g<<5)+(b<<10))

//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;

/////////////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
/////////////////////////////////////////////////////////////////
int main(void)
{
    int x, y;
    unsigned short color;

    SetMode(MODE_3 | BG2_ENABLE);

    while(1)
    {
        //randomize the pixel
        x = rand() % 240;
        y = rand() % 160;
        color = RGB(rand()%31, rand()%31, rand()%31);

        DrawPixel3(x, y, color);
    }

    return 0;
}

//draw a pixel
void DrawPixel3(int x, int y, unsigned short c)
{
    videoBuffer[y * 240 + x] = c;
}


```

```
g:/ham/gcc-arm/bin/arm-thumb-elf-ld.exe    -L g:/ham/gcc-arm/lib/gcc-lib/arm-thumb-elf/3.2.2/normal -L g:/ham/gcc-arm/arm-thumb-elf/lib/normal -L g:/ham/gcc-arm,
g:/ham/gcc-arm/bin/arm-thumb-elf-objcopy.exe    -v -O binary Mode3Pixels.elf Mode3Pixels.gba
copy from Mode3Pixels.elf(elf32-littlearm) to Mode3Pixels.gba(binary)
g:/ham/tools/win32/rm -f *.o *.i *.ii
g:/ham/tools/win32/gbafix.exe   Mode3Pixels.gba
ROM fixed!
g:/ham/tools/win32/vba.exe   Mod
```

Figure 5.3

The Mode3Pixels program draws random pixels on the GBA screen.

Drawing Lines

One of the most commonly requested functions for any computing platform is an algorithm for drawing lines. Most of us would prefer to have an SDK that has hardware support for line drawing. It would be great if the GBA had such a routine built into the hardware, because that would be extremely fast and would allow us to write high-speed polygon-type games (which are based on triangles, which are created with a high-speed line-drawing function). There are several line-drawing algorithms out there, but by far the most common is Bresenham's line drawing algorithm. Since this isn't a book about computer science theory per se, I'm not going to provide a detailed overview of the theory behind this algorithm. It is enough to just use it and assume that it works—by watching the results. The Mode3Lines program demonstrates how to use the DrawPixel3 function to do something more useful than pixel plotting. This project is located on the CD-ROM under \Sources\Chapter05\Mode3Lines.

```
/////////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 5: Bitmap-Based Video Modes
// Mode3Lines Project
// main.c source code file
/////////////////////////////////////////////////////////////

#define MULTIBOOT int __gba_multiboot;
MULTIBOOT

//add support for the rand function
#include <stdlib.h>

//declare the function prototype
void DrawPixel3(int, int, unsigned short);
void DrawLine3(int, int, int, int, unsigned short);

//declare some defines for the video mode
#define REG_DISPCNT *(unsigned long*)0x4000000
#define MODE_3 0x3
#define BG2_ENABLE 0x400

//changes the video mode
#define SetMode(mode) REG_DISPCNT = (mode)

//packs three values into a 15-bit color
```

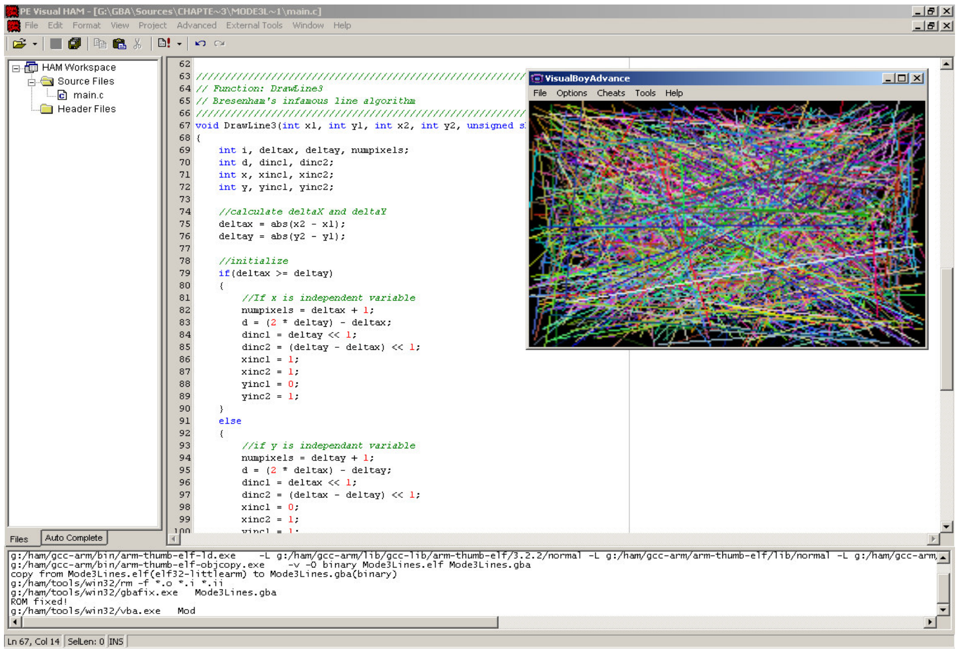
```
#define RGB(r,g,b) ((r)+(g<<5)+(b<<10))

//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;

//////////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
//////////////////////////////////////////////////////////////
int main(void)
{
    int x1,y1,x2,y2;
    unsigned short color;

    SetMode(MODE_3 | BG2_ENABLE);

   while(1)
   {
         x1 = rand() % 240;
        y1 = rand() % 160;
        x2 = rand() % 240;
        y2 = rand() % 160;
        color = RGB(rand()%31, rand()%31, rand()%31);

         DrawLine3(x1,y1,x2,y2,color);
   }

   return 0;
}

//////////////////////////////////////////////////////////////
// Function: DrawPixel3
// Draws a pixel in mode 3
//////////////////////////////////////////////////////////////
```

```
void DrawPixel3(int x, int y, unsigned short color)
{
    videoBuffer[y * 240 + x] = color;
}

/////////////////////////////////////////////////////////////
// Function: DrawLine3
// Bresenham's infamous line algorithm
/////////////////////////////////////////////////////////////
void DrawLine3(int x1, int y1, int x2, int y2, unsigned short color)
{
    int i, deltax, deltay, numpixels;
    int d, dinc1, dinc2;
    int x, xinc1, xinc2;
    int y, yinc1, yinc2;

    //calculate deltaX and deltaY
    deltax = abs(x2 - x1);
    deltay = abs(y2 - y1);

    //initialize
    if(deltax >= deltay)
    {
        //If x is independent variable
        numpixels = deltax + 1;
        d = (2 * deltay) - deltax;
        dinc1 = deltay << 1;
        dinc2 = (deltay - deltax) << 1;
        xinc1 = 1;
        xinc2 = 1;
        yinc1 = 0;
        yinc2 = 1;
    }
    else
```

```
{
    //if y is independent variable
    numpixels = deltay + 1;
    d = (2 * deltax) - deltay;
    dinc1 = deltax << 1;
    dinc2 = (deltax - deltay) << 1;
    xinc1 = 0;
    xinc2 = 1;
    yinc1 = 1;
    yinc2 = 1;
}

//move the right direction
if(x1 > x2)
{
    xinc1 = -xinc1;
    xinc2 = -xinc2;
}
if(y1 > y2)
{
    yinc1 = -yinc1;
    yinc2 = -yinc2;
}

x = x1;
y = y1;

//draw the pixels
for(i = 1; i < numpixels; i++)
{
    DrawPixel3(x, y, color);

    if(d < 0)
    {
```

```
                d = d + dinc1;
                x = x + xinc1;
                y = y + yinc1;
            }
            else
            {
                d = d + dinc2;
                x = x + xinc2;
                y = y + yinc2;
            }
        }
    }
```

The output from the Mode3Lines program is shown in Figure 5.4.


PE Visual HAM - [G:\GBA\Sources\CHAPTE~3\MODE3L~1\main.c]
File Edit Format View Project Advanced External Tools Window Help
HAM Workspace
Source Files
main.c
Header Files
Files
Auto Complete
VisualBoyAdvance
File Options Cheats Tools Help
Ln 67, Col 14
SelLen: 0
INS

```
//////////////////////////////////////////////////////////////
// Function: DrawLine3
// Bresenham's infamous line algorithm
//////////////////////////////////////////////////////////////
void DrawLine3(int x1, int y1, int x2, int y2, unsigned s
{
    int i, deltax, deltay, numpixels;
    int d, dinc1, dinc2;
    int x, xinc1, xinc2;
    int y, yinc1, yinc2;

    //calculate deltaX and deltaY
    deltax = abs(x2 - x1);
    deltay = abs(y2 - y1);

    //initialize
    if(deltax >= deltay)
    {
        //If x is independent variable
        numpixels = deltax + 1;
        d = (2 * deltay) - deltax;
        dinc1 = deltay << 1;
        dinc2 = (deltay - deltax) << 1;
        xinc1 = 1;
        xinc2 = 1;
        yinc1 = 0;
        yinc2 = 1;
    }
    else
    {
        //if y is independant variable
        numpixels = deltay + 1;
        d = (2 * deltax) - deltay;
        dinc1 = deltax << 1;
        dinc2 = (deltax - deltay) << 1;
        xinc1 = 0;
        xinc2 = 1;
        yinc1 = 1;
```

```
g:/ham/gcc-arm/bin/arm-thumb-elf-ld.exe    -L g:/ham/gcc-arm/lib/gcc-lib/arm-thumb-elf/3.2.2/normal -L g:/ham/gcc-arm/arm-thumb-elf/lib/normal -L g:/ham/gcc-arm
g:/ham/gcc-arm/bin/arm-thumb-elf-objcopy.exe    -v -O binary Mode3Lines.elf Mode3Lines.gba
copy from Mode3Lines.elf(elf32-littlearm) to Mode3Lines.gba(binary)
g:/ham/tools/win32/rm -f *.o *.i *.ii
g:/ham/tools/win32/gbafix.exe   Mode3Lines.gba
ROM fixed!
g:/ham/tools/win32/vba.exe   Mod
```

Figure 5.4

The Mode3Lines program draws random lines on the GBA screen.

Making GBA Programs Multi-Bootable

At the top of the code listing for the Mode3Lines program is a rather strange-looking couple of lines:

```
#define MULTIBOOT int __gba_multiboot;
MULTIBOOT
```

These two lines of code are very important, so I will mention them in each chapter from this point forward (since many readers prefer to jump to their favorite chapters, and may not necessarily read through the whole book). Multi-boot is a feature of the GBA that allows it to run multi-player games using link cables, where a single GBA has a multi-player game cartridge, while the other players may go without. For example, *The Legend of Zelda: A Link To The Past* includes a four-player mini-game called *Four Swords* that can be played with two, three, or four players.

To run a multi-boot game, simply plug in the link cables, remove any cartridges from the client players, and leave in the game cartridge for the host player. When the GBA detects that there is no cartirdge, it goes into multi-player mode, and attempts to download a ROM over the link cable. The Multi-Boot Version 2 (MBV2) cable, available from http://www.lik-sang.com, takes advantage of this feature by allowing you to test your GBA programs using a special utility program that sends a ROM through the PC's parallel port to the GBA, in the same manner that a multi-player game ROM is transferred from a host GBA to client players.

The definition

```
#define MULTIBOOT int __gba_multiboot;
```

is simply a compiler directive that creates an int variable called __gba_multiboot (note the double underscores in front, and single underscore between gba and multiboot). The HAM SDK recognizes this variable declaration as an instruction to make the ROM image multi-bootable, which adds support for MBV2.

However, just creating the definition using #define doesn't make a game bootable. Rather, you must use the definition, which is what the second line does:

```
MULTIBOOT
```

If you prefer, you may simply insert a single line at the top of the program like this:

```
int __gba_multiboot;
```

and that will suffice. However, everyone in the GBA community is using MULTIBOOT, and it is a standard definition in the popular header files (such as gba.h and mygba.h, included with many public domain GBA games).

The only condition on using MULTIBOOT is that it must be included at the top of the source code listing, even before the include statements, and the ROM may not be larger than 256 KB. Unfortunately, that is the limit of the RAM on the GBA. Remember, there is no cartridge in the GBA, with multiple megabytes of space available, just the RAM! If you are working on a sizeable game, the MBV2 may not work for a project if the ROM is too big.

However, the sample programs in this book are all fairly small, so each sample program from this point forward will include the MULTIBOOT statement. If you would like to test a multi-boot program yourself, you will first need the MBV2 device. It is around $30.00, so if you plan to do a lot of GBA development, I highly recommend buying one from Lik-Sang or another distributor. If you do have a MBV2, and want to try it out immediately, I have included a folder on the CD-ROM called \MultiBoot. This folder includes a compiled version of every sample program in the book, ready to be run via MBV2. The MB.EXE is also in the folder, along with a convenient batch file called multiboot.bat that uses the appropriate options.

Note that an extra step is required for Windows 2000 or XP. First, you must install the userport driver, due to the way the MBV2 adapter works. I have included the userport driver in the \MultiBoot folder under \MultiBoot\Win_2K_XP. First, copy the userport.sys file to your \WINNT\SYSTEM32\DRIVERS folder. Next, run userport.exe to configure the parallel port as appropriate for your PC. Normal configurations will use LPT1 at address 378-37F. You may read the readme.txt file or the userport.pdf file (using Adobe Acrobat Reader) for more help with the userport driver. Once you have run the userport.exe file and configured the port address, you may then run the MB.EXE program to transfer a program to the GBA and watch it run on the actual hardware. Depending on your system, you may need to run the userport.exe program each time you start using the MBV2 (on a per-boot basis, or each time you open the Command Prompt, depending on the version of Windows you are using—and note that userport is not needed for Windows 98/ME).

To run a program on your GBA, first remove any cartridge, then turn on your GBA. The Game Boy logo will appear, and then it will wait for a signal from MBV2. Now, open a Command Prompt (Start, Run, cmd.exe). CD to the \MultiBoot folder on the CD-ROM (or copy the folder to your hard drive first, and CD to that folder), then type

```
multiboot Mode3Lines
```

without the extension, because .gba is added to the filename by the batch file. For reference, the multiboot.bat file includes the following command:

```
mb -w 50 -s %1.gba
```

Note that I have tested all of the ROMs in the \MultiBoot folder on an actual GBA, using the MBV2 cable, and verified that they all work, at least with the configuration on my development PC (which I have been describing here). Some of the sample programs look amazing running on the GBA! For instance, the sample programs from Chapter 7, "Rounding Up Sprites." While I wouldn't encourage you to skip ahead so soon, the sprite demos are particularly impressive, such as TransSprite, which displays five bouncing balls that alternate between solid and translucent, using alpha blending! However, the sample programs from this chapter and all of them, really, are loads of fun to demo on an actual GBA. If you don't own a MBV2, I apologize for in After running the command, you should see a message that looks something like this:

```
Parallel MBV2 cable found: hardware=v2.0, firmware=v4

EPP support found.

Looking for multiboot-ready GBA...

Sending data. Please wait...

CRC Ok - Done.
```

If the GBA is not turned on, or if the parallel port has not been configured properly, you may get an error message such as the following.

```
ERROR - No hardware found on selected parallel port!

Check parallel port number, make sure cable is connected to port and to GBA,

& make sure GBA is turned on.
```

I recommend setting it to an Enhanced Parallel Port (EPP) in your PC's BIOS setup. If your PC doesn't have a parallel port with EPP mode, that's okay, the MBV2 will still work fine on it, but you may have to adjust the wait period. I have used 50 milliseconds, because that is the value that worked best for my PC. If you keep getting timeout errors, you might bump the wait to -w 100, although if that still doesn't work, I would suspect a problem with your ports or the userport driver.

Running the sample programs through the MBV2 directly on your GBA is a fascinating experience. Watching your own games run on the actual handheld console is nothing short of thrilling, and you will amaze your friends and relatives by showing them your game running on an actual GBA! That is why I highly recommend buying a MBV2 adapter. They are $30.00, and only available via mail order. Shipping is rather high from Asia, so you may want to save up and pick up a few things to combine shipping (for instance, order a Flash Advance Linker at the same time). Even with a Flash Advance Linker, I highly recommend a MBV2, simply because it's fast and easy to use. You can compile and run programs via MBV2 directly from within Visual HAM (see the Projects menu). Sometimes it can be a little frustrating to get the MBV2 to work flawlessly every single time. If you are having trouble getting it to work, I recommend you check the MBV2 FAQ at http://www.devrs.com/gba/files/mbv2faqs.php. Of course, you may also browse for "MBV2" on Google.com or another search engine to find additional resources.

Drawing Circles

Bresenham was a genius, because he not only provided us with a cool line-drawing algorithm, but he also created a great circle-drawing algorithm too! As with the line algorithm, if you want the theory behind it, I would recommend picking up a book on graphics theory or searching the Web for a suitable tutorial. The Mode3Circles program demonstrates how to use the Bresenham algorithm for drawing circles in mode 3.

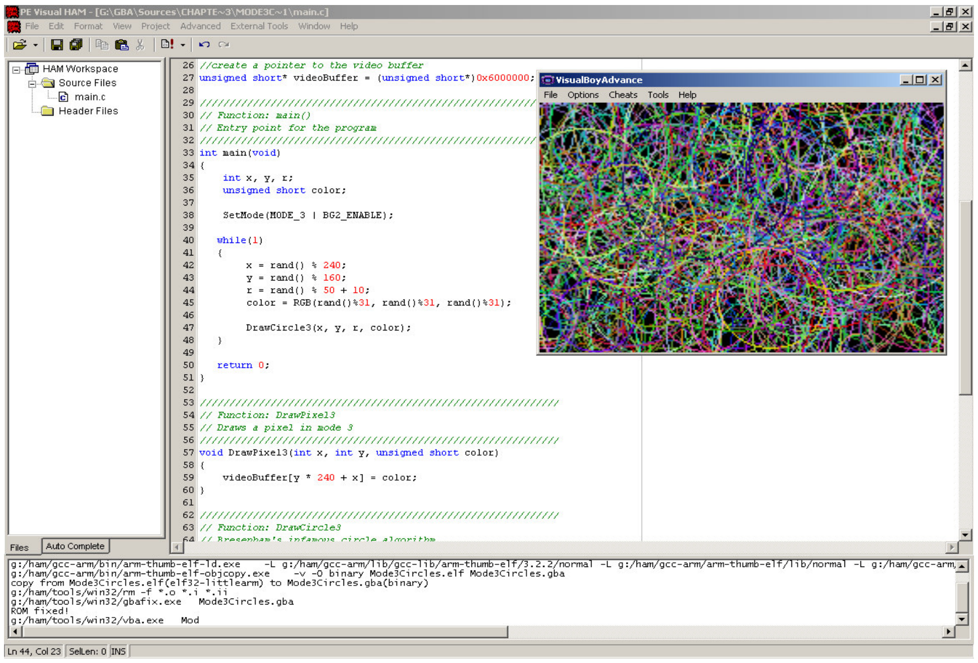
```
/////////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 5: Bitmap-Based Video Modes
// Mode3Circles Project
// main.c source code file
/////////////////////////////////////////////////////////////

#define MULTIBOOT int __gba_multiboot;
MULTIBOOT

//add support for the rand function
#include <stdlib.h>

//declare the function prototype
void DrawPixel3(int, int, unsigned short);
void DrawCircle3(int, int, int, int);

//declare some defines for the video mode
#define REG_DISPCNT *(unsigned long*)0x4000000
```

```
#define MODE_3 0x3
#define BG2_ENABLE 0x400

//changes the video mode
#define SetMode(mode) REG_DISPCNT = (mode)

//packs three values into a 15-bit color
#define RGB(r,g,b) ((r)+(g<<5)+(b<<10))

//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;

/////////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
/////////////////////////////////////////////////////////////
int main(void)
{
    int x, y, r;
    unsigned short color;

    SetMode(MODE_3 | BG2_ENABLE);

   while(1)
   {
        x = rand() % 240;
        y = rand() % 160;
        r = rand() % 50 + 10;
        color = RGB(rand()%31, rand()%31, rand()%31);

         DrawCircle3(x, y, r, color);
   }

   return 0;
```

```
}

//////////////////////////////////////////////////////////////
// Function: DrawPixel3
// Draws a pixel in mode 3
//////////////////////////////////////////////////////////////
void DrawPixel3(int x, int y, unsigned short color)
{
     videoBuffer[y * 240 + x] = color;
}

//////////////////////////////////////////////////////////////
// Function: DrawCircle3
// Bresenham's infamous circle algorithm
//////////////////////////////////////////////////////////////
void DrawCircle3(int xCenter, int yCenter, int radius, int color)
{
     int x = 0;
     int y = radius;
     int p = 3 - 2 * radius;
     while (x <= y)
     {
          DrawPixel3(xCenter + x, yCenter + y, color);
          DrawPixel3(xCenter - x, yCenter + y, color);
          DrawPixel3(xCenter + x, yCenter - y, color);
          DrawPixel3(xCenter - x, yCenter - y, color);
          DrawPixel3(xCenter + y, yCenter + x, color);
          DrawPixel3(xCenter - y, yCenter + x, color);
          DrawPixel3(xCenter + y, yCenter - x, color);
          DrawPixel3(xCenter - y, yCenter - x, color);

          if (p < 0)
               p += 4 * x++ + 6;
          else
```

```
                p += 4 * (x++ - y--) + 10;
        }
}
```

The output from the Mode3Circles program is shown in Figure 5.5.


PE Visual HAM - [G:\GBA\Sources\CHAPTE~3\MODE3C~1\main.c]
File Edit Format View Project Advanced External Tools Window Help
HAM Workspace
Source Files
main.c
Header Files
Files
Auto Complete
VisualBoyAdvance
File Options Cheats Tools Help
Ln 44, Col 23 SelLen: 0 INS

```
//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;

/////////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
/////////////////////////////////////////////////////////////
int main(void)
{
    int x, y, r;
    unsigned short color;

    SetMode(MODE_3 | BG2_ENABLE);

   while(1)
   {
        x = rand() % 240;
        y = rand() % 160;
        r = rand() % 50 + 10;
        color = RGB(rand()%31, rand()%31, rand()%31);

        DrawCircle3(x, y, r, color);
   }

   return 0;
}

/////////////////////////////////////////////////////////////
// Function: DrawPixel3
// Draws a pixel in mode 3
/////////////////////////////////////////////////////////////
void DrawPixel3(int x, int y, unsigned short color)
{
    videoBuffer[y * 240 + x] = color;
}

/////////////////////////////////////////////////////////////
// Function: DrawCircle3
// Bresenham's infamous circle algorithm
```

```
g:/ham/gcc-arm/bin/arm-thumb-elf-ld.exe    -L g:/ham/gcc-arm/lib/gcc-lib/arm-thumb-elf/3.2.2/normal -L g:/ham/gcc-arm/arm-thumb-elf/lib/normal -L g:/ham/gcc-arm
g:/ham/gcc-arm/bin/arm-thumb-elf-objcopy.exe    -v -O binary Mode3Circles.elf Mode3Circles.gba
copy from Mode3Circles.elf(elf32-littlearm) to Mode3Circles.gba(binary)
g:/ham/tools/win32/rm -f *.o *.i *.ii
g:/ham/tools/win32/gbafix.exe   Mode3Circles.gba
ROM fixed!
g:/ham/tools/win32/vba.exe   Mod
```

Figure 5.5
The Mode3Circles program draws random circles on the GBA screen.

Drawing Filled Boxes

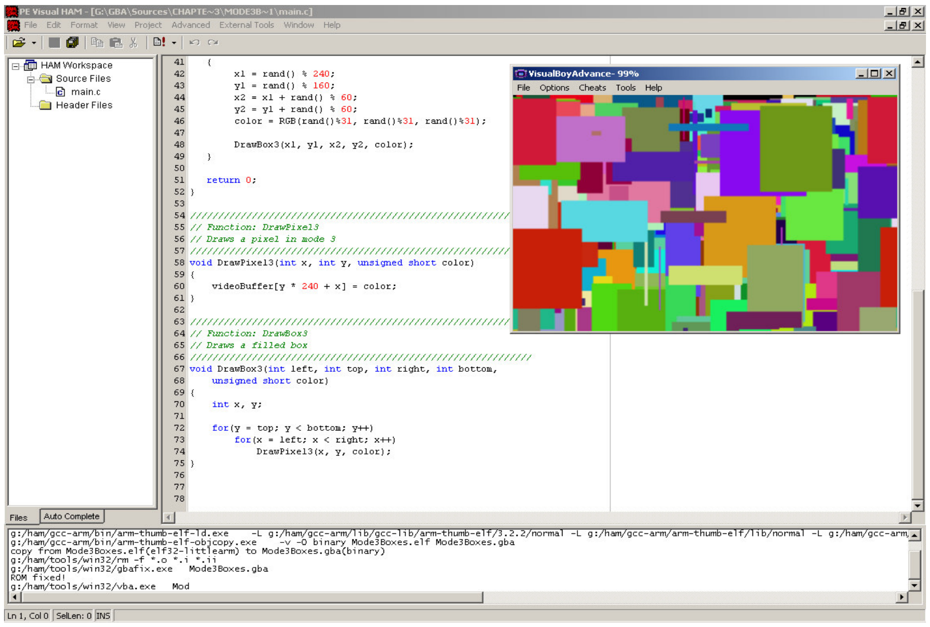
How about something a little more interesting? It's truly amazing what you can do after you have the basic pixel-plotting code available! The Mode3Boxes program draws filled boxes on the screen. While I could have used Bresenham's line algorithm, that would have been grossly wasteful, because that algorithm is only useful for diagonal lines. When it comes to straight lines in a filled box, all you need to do is throw in a couple of loops and draw the pixels. This project is called Mode3Boxes and is located on the CD-ROM under \Sources\Chapter05\Mode3Boxes.

```
/////////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 5: Bitmap-Based Video Modes
// Mode3Boxes Project
// main.c source code file
/////////////////////////////////////////////////////////////
```

```
#define MULTIBOOT int __gba_multiboot;
MULTIBOOT

//add support for the rand function
#include <stdlib.h>

//declare the function prototype
void DrawPixel3(int, int, unsigned short);
void DrawBox3(int, int, int, int, unsigned short);

//declare some defines for the video mode
#define REG_DISPCNT *(unsigned long*)0x4000000
#define MODE_3 0x3
#define BG2_ENABLE 0x400

//changes the video mode
#define SetMode(mode) REG_DISPCNT = (mode)

//packs three values into a 15-bit color
#define RGB(r,g,b) ((r)+(g<<5)+(b<<10))

//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;

/////////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
/////////////////////////////////////////////////////////////
int main(void)
{
    int x1, y1, x2, y2;
    unsigned short color;

    SetMode(MODE_3 | BG2_ENABLE);
   while(1)
   {
         x1 = rand() % 240;
        y1 = rand() % 160;
         x2 = x1 + rand() % 60;
        y2 = y1 + rand() % 60;
        color = RGB(rand()%31, rand()%31, rand()%31);

         DrawBox3(x1, y1, x2, y2, color);
   }

   return 0;
}

/////////////////////////////////////////////////////////////
// Function: DrawPixel3
// Draws a pixel in mode 3
/////////////////////////////////////////////////////////////
void DrawPixel3(int x, int y, unsigned short color)
{
     videoBuffer[y * 240 + x] = color;
}

/////////////////////////////////////////////////////////////
// Function: DrawBox3
// Draws a filled box
/////////////////////////////////////////////////////////////
void DrawBox3(int left, int top, int right, int bottom,
    unsigned short color)
{
     int x, y;

    for(y = top; y < bottom; y++)
```

```
        for(x = left; x < right; x++)

            DrawPixel3(x, y, color);

}
```

The output from the Mode3Boxes program is shown in Figure 5.6.



```
    {
        x1 = rand() % 240;
        y1 = rand() % 160;
        x2 = x1 + rand() % 60;
        y2 = y1 + rand() % 60;
        color = RGB(rand()%31, rand()%31, rand()%31);

        DrawBox3(x1, y1, x2, y2, color);
    }

    return 0;
}

//////////////////////////////////////////////////////////////
// Function: DrawPixel3
// Draws a pixel in mode 3
//////////////////////////////////////////////////////////////
void DrawPixel3(int x, int y, unsigned short color)
{
    videoBuffer[y * 240 + x] = color;
}

//////////////////////////////////////////////////////////////
// Function: DrawBox3
// Draws a filled box
//////////////////////////////////////////////////////////////
void DrawBox3(int left, int top, int right, int bottom,
    unsigned short color)
{
    int x, y;

    for(y = top; y < bottom; y++)
        for(x = left; x < right; x++)
            DrawPixel3(x, y, color);
}
```

```
g:/ham/gcc-arm/bin/arm-thumb-elf-ld.exe    -L g:/ham/gcc-arm/lib/gcc-lib/arm-thumb-elf/3.2.2/normal -L g:/ham/gcc-arm/arm-thumb-elf/lib/normal -L g:/ham/gcc-arm
g:/ham/gcc-arm/bin/arm-thumb-elf-objcopy.exe    -v -O binary Mode3Boxes.elf Mode3Boxes.gba
copy from Mode3Boxes.elf(elf32-littlearm) to Mode3Boxes.gba(binary)
g:/ham/tools/win32/rm -f *.o *.i *.ii
g:/ham/tools/win32/gbafix.exe   Mode3Boxes.gba
ROM fixed!
g:/ham/tools/win32/vba.exe   Mod
```

PE Visual HAM - [G:\GBA\Sources\CHAPTE~3\MODE3B~1\main.c]
File Edit Format View Project Advanced External Tools Window Help
HAM Workspace
Source Files
main.c
Header Files
Files
Auto Complete
VisualBoyAdvance- 99%
File Options Cheats Tools Help
Ln 1, Col 0 SelLen: 0 INS

Figure 5.6

The Mode3Boxes program draws random filled boxes on the GBA screen.

Drawing Bitmaps

Now for something really interesting (as if the last few sections haven't already been?). When it comes down to it, you can make a game using just pixels, lines, and other vector objects. But when you want to do something really cool, you need bitmaps. I'm not covering sprites until later (Chapter 7), but for now I'd like to introduce you to bitmap images and show you how to draw them on the screen.

Converting 8-Bit Bitmaps to 15-Bit Images

The first thing you need to do is create the source bitmap to display and tweak the color depth so it is saved in 8-bit, 256-color mode. This does reduce the number of possible colors in 15-bit mode 3 and mode 5, but the gfx2gba program can only read 8-bit images (and since it is the most popular tool, we'll just work around the limitation). Note that the GBA only uses 15 out of 16 bits in a 555 format (that's Blue/Green/Red, or more commonly referred to as BGR), and the last bit is ignored.

There is a good bonus to using 8-bit images for all of your source artwork, because then you need to only keep track of a single collection of artwork and need not worry about keeping two versions (both 8-bit and 16-bit image files).

Converting Bitmap Images to Game Boy Format

The GBA doesn't have a file system of any kind, so you must embed the bitmaps (as well as sound effects, music, and any other data files) into source code as an array. The nice thing about this is that you really don't have to worry about writing code to load a bitmap, sound, or other resource, because it's immediately available to your program as a C-style array—such as the one I showed you at the beginning of this chapter. Several tools are available in the public domain to convert a bitmap file to a GBA source code file; the most common of these is gfx2gba.exe. There is no formal support or installer for this tool—it's just a public domain program (like most of the utility programs for the GBA). The gfx2gba.exe program is automatically installed with HAM, so there is no need to download it. The file is located in the \ham\tools\win32 folder, on whatever drive you installed HAM to.

A convenient batch file included with HAM called startham.bat sets up a path to this folder so you can run the utility programs from the command-line prompt. Just open a command window by clicking Start, Run and typing in "cmd". Then click on the OK button. A command window should open, providing a prompt that is familiar to those of us who once used MS-DOS. You will need to change to the folder where your bitmap file is located in order to convert it. Of course, you can also type in a fully qualified pathname to your source bitmap, but I prefer to run gfx2gba.exe while in the source folder already. You can move to the folder for the next sample project in this chapter by typing "CD \Sources\Chapter05\Mode3Bitmap". I am assuming here that you have copied the \Sources folder off the CD-ROM to the root of your hard drive. Of course, you will want to modify that pathname to match the folder you are using for book projects.

To convert a bitmap using gfx2gba.exe for mode 3 (which is not palettized), you will want to type in this command:

```
gfx2gba -fsrc -c32k filename.bmp
```

The source file could also be a .pcx image. The important thing is that the image must be saved in 8-bit, 256-color mode, because that is the only format that gfx2gba supports. Now for a short explanation of the options used. The -fsrc option specifies that the output should be C source code. The -c32k option tells gfx2gba to output nonpalettized 15-bit pixels (without this option, only 8-bit palettized colors are used, which won't work in mode

3 or mode 5). Finally, the last option specifies the source file to convert. Now that you know this, I can show you the command to convert the mode3.bmp file for the upcoming project:

```
gfx2gba -fsrc -c32k mode3.bmp
```

This command produces one output file called mode3.raw.c. The mode3.raw.c file looks similar to the file I showed you earlier:

```
//
// mode3 (76800 uncompressed bytes)
//
const unsigned short mode3_Bitmap[38400] = {
0x7717, 0x7293, 0x69cd, 0x6e0f, 0x6e0f, 0x6e0f, 0x6e0f, 0x6e0f,
0x6e0f, 0x6e0f, 0x6e0f, 0x6e0f, 0x6e0f, 0x6e0f, 0x6e0f, 0x6e0f,
0x6e0f, 0x6e0f, 0x7293, 0x7717, 0x6e51, 0x6e0f, 0x7717, 0x6e51,
0x656a, 0x69cd, 0x61aa, 0x58c3, 0x6e51, 0x6e51, 0x7293, 0x7717,
0x7bbc, 0x7b5a, 0x6a2f, 0x6a2f, 0x6e51, 0x69cd, 0x656a, 0x6a2f,
0x6a2f, 0x6a2f, 0x6a2f, 0x6a2f, 0x6a2f, 0x6a2f, 0x7717, 0x7b9b,
0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7759, 0x7293,
0x6e51, 0x6a2e, 0x6a2f, 0x6a72, 0x7293, 0x6127, 0x58c3, 0x58c3,
 . . .
```

Note that all full-size 15-bit bitmaps will be 76,800 bytes in length, while the file I showed you earlier was a 38,400-byte image—which was a palettized image for mode 4. The palette file is something you have not seen yet, but I will cover that in the section on mode 4.

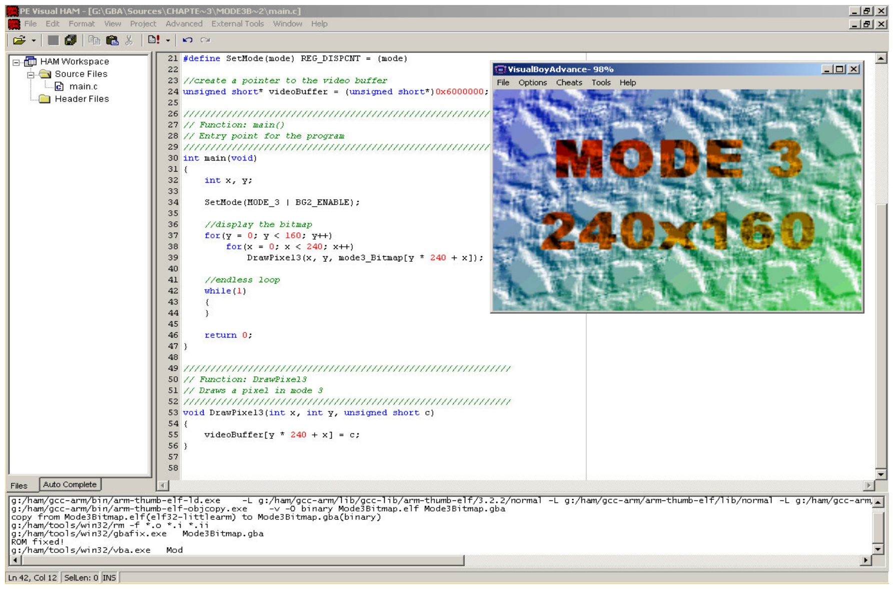
Drawing Converted Bitmaps

After you have converted a bitmap file to C source, you can then directly use the image in a GBA program without any real work on your part. That is one nice aspect of GBA graphics programming—all the work is done up front. Here's the source code for the Mode3Bitmap program:

```
//////////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 5: Bitmap-Based Video Modes
// Mode3Bitmap Project
// main.c source code file
//////////////////////////////////////////////////////////////

#define MULTIBOOT int __gba_multiboot;
MULTIBOOT

//include files
#include <stdlib.h>
#include "mode3.raw.c"

//declare the function prototype
void DrawPixel3(int, int, unsigned short);

//declare some defines for the video mode
#define REG_DISPCNT *(unsigned long*)0x4000000
#define MODE_3 0x3
#define BG2_ENABLE 0x400

//changes the video mode
#define SetMode(mode) REG_DISPCNT = (mode)

//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;

//////////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
//////////////////////////////////////////////////////////////
int main(void)
{
    int x, y;

    SetMode(MODE_3 | BG2_ENABLE);
    //display the bitmap
    for(y = 0; y < 160; y++)
        for(x = 0; x < 240; x++)
            DrawPixel3(x, y, mode3_Bitmap[y * 240 + x]);

    //endless loop
    while(1)
    {
    }

    return 0;
}

/////////////////////////////////////////////////////////////
// Function: DrawPixel3
// Draws a pixel in mode 3
/////////////////////////////////////////////////////////////
void DrawPixel3(int x, int y, unsigned short c)
{
    videoBuffer[y * 240 + x] = c;
}
```

The output from the Mode3Bitmap program is shown in Figure 5.7.

There's a faster way to draw bitmaps using DMA, due to a high-speed hardware-based memory copy feature. Since it's based in hardware, you can expect it to be several times faster than displaying the bitmap one pixel at a time. I will cover DMA features in Chapter 8, "Using Interrupts and Timers."


PE Visual HAM - [G:\GBA\Sources\CHAPTE~3\MODE3B~2\main.c]
File Edit Format View Project Advanced External Tools Window Help
HAM Workspace
Source Files
main.c
Header Files
Files
Auto Complete
VisualBoyAdvance- 98%
File Options Cheats Tools Help
MODE 3
240x160
Ln 42, Col 12 SelLen: 0 INS

```
#define SetMode(mode) REG_DISPCNT = (mode)

//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;

/////////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
/////////////////////////////////////////////////////////////
int main(void)
{
    int x, y;

    SetMode(MODE_3 | BG2_ENABLE);

    //display the bitmap
    for(y = 0; y < 160; y++)
        for(x = 0; x < 240; x++)
            DrawPixel3(x, y, mode3_Bitmap[y * 240 + x]);

    //endless loop
    while(1)
    {
    }

    return 0;
}

///////////////////////////////////////////////////////////////
// Function: DrawPixel3
// Draws a pixel in mode 3
///////////////////////////////////////////////////////////////
void DrawPixel3(int x, int y, unsigned short c)
{
    videoBuffer[y * 240 + x] = c;
}
```

```
g:/ham/gcc-arm/bin/arm-thumb-elf-ld.exe    -L g:/ham/gcc-arm/lib/gcc-lib/arm-thumb-elf/3.2.2/normal -L g:/ham/gcc-arm/arm-thumb-elf/lib/normal -L g:/ham/gcc-arm,
g:/ham/gcc-arm/bin/arm-thumb-elf-objcopy.exe    -v -O binary Mode3Bitmap.elf Mode3Bitmap.gba
copy from Mode3Bitmap.elf(elf32-littlearm) to Mode3Bitmap.gba(binary)
g:/ham/tools/win32/rm -f *.o *.i *.ii
g:/ham/tools/win32/gbafix.exe   Mode3Bitmap.gba
ROM fixed!
g:/ham/tools/win32/vba.exe   Mod
```

Figure 5.7

The Mode3Bitmap program draws a bitmap image on the GBA screen.

Working with Mode 4

Video mode 4 is an 8-bit (palettized) 240 x 160 display with a more difficult pixel format where two pixels are packed into a 16-bit value, and there is no other way to read or write values in this memory buffer except 16 bits at a time (which is an unsigned short value).

Dealing with Palettes

Mode 4 is an 8-bit palettized display mode, which means that there are only 256 total colors available, and each color is stored in a lookup table called the *palette*. Here is a sample mode 4 palette output by the gfx2gba program:

```
const unsigned short mode4_Palette[256] = {
0x0001, 0x0023, 0x0026, 0x0086, 0x002a, 0x008a, 0x00aa, 0x010a,
0x00a9, 0x00ad, 0x00ae, 0x014e, 0x3940, 0x190b, 0x44e0, 0x4522,
0x25c0, 0x1e20, 0x3181, 0x2dc1, 0x2a23, 0x15cb, 0x3604, 0x3629,
0x4183, 0x45a5, 0x41e5, 0x4206, 0x3e27, 0x4627, 0x4228, 0x420c,
0x5481, 0x58c3, 0x5525, 0x6127, 0x4d86, 0x4dc7, 0x5186, 0x5988,
.
.
.
0x5fb5, 0x6357, 0x6396, 0x63b7, 0x639a, 0x6b36, 0x6b57, 0x6f37,
0x7357, 0x6f58, 0x7359, 0x7759, 0x7b5a, 0x6b78, 0x6f99, 0x6b99,
0x6fba, 0x739a, 0x7779, 0x7b9a, 0x6fdb, 0x73dc, 0x77bb, 0x7b9b,
0x7bbc, 0x7bdc, 0x7bdd, 0x77fd, 0x7fde, 0x7bfe, 0x7ffe, 0x77bf,
0x7bdf, 0x7fff, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000
};
```

The palette file isn't nearly as long as the raw file, but I have truncated it to save space. You may open the files yourself using Notepad or Visual HAM to see the full listing. When working with mode 4, you need to set up the palette table first before drawing anything, because most likely the palette entries are all 0.

The palette memory on the GBA is a 256-byte section of memory located at memory address 0 x 5000000. To use the palette, then, you'll need to create a pointer to this location in memory:

```
unsigned short* paletteMem = (unsigned short*)0x5000000;
```

Setting the palette entries is then simply a matter of setting the value for each element of the 256 paletteMem array. For instance, this code sets all the colors of the palette to white:

```
for (n = 0; n < 256; n++)
    paletteMem[n] = RGB(31, 31, 31);
```

Most games that use mode 4 have a master palette that is used by all the graphics in the game. The reason this is necessary is because you can only have one palette active at a time. Now, you could easily change the palette for each game level, and that is what most games do! There's no need to limit your creativity just because there are only 256 colors available at a time. By having several (or a dozen) palette arrays available, one per level of the game, your game can be very colorful and each level can be distinctive. Note that it's common practice to keep palette entry 0 set to black (that is, 0 x 00) because that is often used as the color for transparency. Remember, even the sprites in your game will use this palette (unless the sprites each have their own 16-color palette—more on that in Chapter 7, "Rounding Up Sprites"). Figure 5.8 shows an illustration that helps to explain the difference between mode 3 (15-bit) and mode 4 (8-bit).

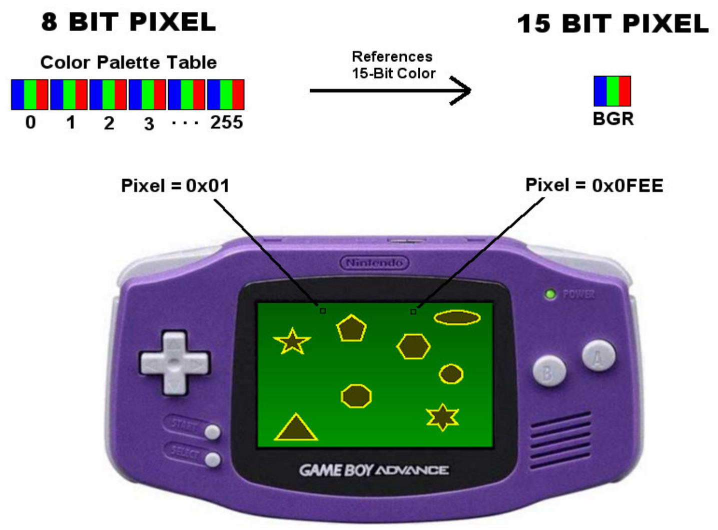


Figure 5.8

Comparison of 8-bit and 15-bit display modes.

Drawing Pixels

As I mentioned briefly a moment ago, mode 4 uses a more difficult pixel-packing method than the straightforward method used in modes 3 and 5 (where each pixel is represented by a single unsigned short). The reason why mode 4 is more difficult is because it's an 8-bit mode, and the video display system on the GBA can only handle 16-bit values. The video chip literally blasts two pixels at a time on the LCD screen, which is great for speed; however, the drawback is that setting pixels is something of a bottleneck. Basically, each even pixel is stored in the lower half of the unsigned short value, while each odd pixel is stored in the upper half. Figure 5.9 illustrates the point.

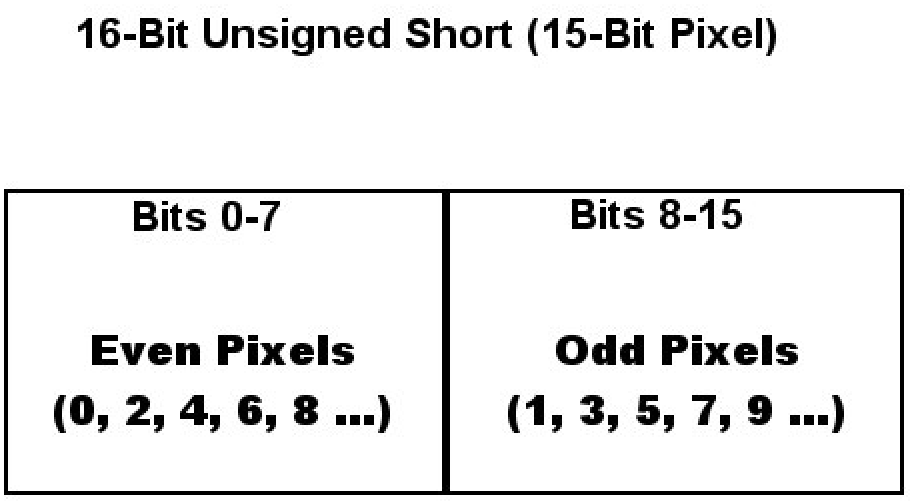


Figure 5.9

Mode 4 pixels are packed in twos for every 16-bit number in the video buffer.

In order to set a pixel, one must first read the existing 16-bit value, combine it with the new pixel value, and then write it back to the video buffer. As you can deduce, three steps are essentially required to set a pixel instead of just one step—which is a given when working with transparent sprites. Fortunately, the hardware sprite handler takes care of that, but this support doesn't necessitate ignoring how this video mode works for your own needs, so I'm going to show you how to write a pixel in this mode.

First, read the existing unsigned short value, dividing the x value by 2:

```
unsigned short offset = (y * 240 + x) >> 1;

pixel = videoBuffer[offset];
```

Next, determine whether x is even or odd and AND'ing x with 1:

```
if (x & 1)
```

Finally, if x is even, then copy the pixel to the lower portion of the unsigned short. In order to do this, you must shift the color bits left by 8 bits so they can be combined with a number that is right-aligned, like this:

```
videoBuffer[offset] = (color << 8) + (pixel & 0x00FF);
```

If x is odd, then copy it to the upper portion of the number, without worrying about bit shifting, like so:

```
videoBuffer[offset] = (pixel & 0xFF00) + color;
```

This is obviously slower than simply writing a pixel to the video buffer. However, mode 4 has an advantage of being fast when using hardware-accelerated blitting functions and DMA, because twice as many pixels can be copied to video memory. What mode 4 is doing is essentially packing two pixels into the same space occupied by one pixel in modes 3 and 5. Okay, so let's write a function to plot pixels in mode 4.

```
void DrawPixel4(int x, int y, unsigned char color)
{
    unsigned short pixel;
    unsigned short offset = (y * 240 + x) >> 1;

    pixel = videoBuffer[offset];
    if (x & 1)
        videoBuffer[offset] = (color << 8) + (pixel & 0x00FF);
    else
```

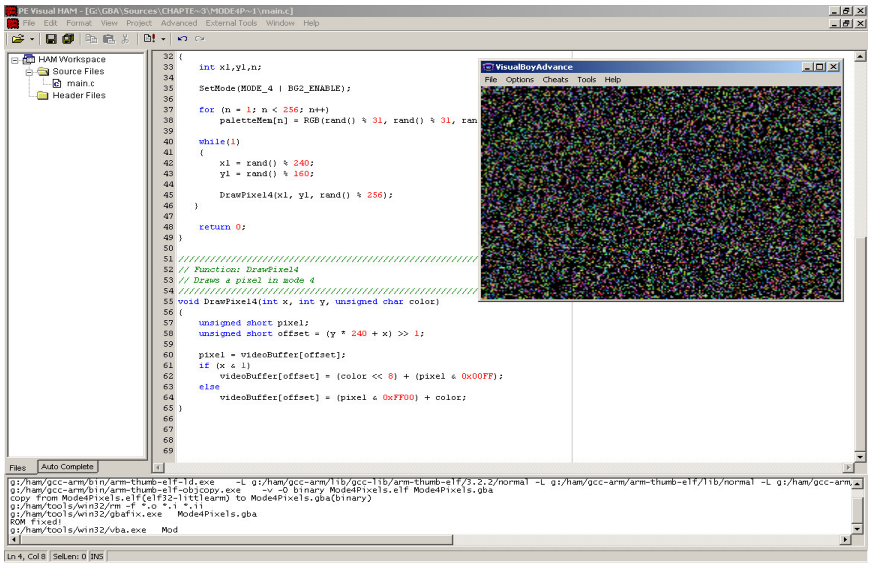
```
        videoBuffer[offset] = (pixel & 0xFF00) + color;
}
```

The Mode4Pixels program uses the standard library (stdlib.h) to gain access to a pseudo-random number generator in order to plot random pixels on the screen. This is the same rand() function you have seen in earlier programs in this chapter. The source code for the Mode4Pixels program should be helpful to you if you ever plan to write a game using mode 4, because the DrawPixel4 function can be very helpful indeed. I somewhat dislike mode 4 because of the packed pixels, but it does have advantages, as I have explained already. Here is the source code for Mode4Pixels.

```
/////////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 5: Bitmap-Based Video Modes
// Mode4Pixels Project
// main.c source code file
/////////////////////////////////////////////////////////////

#define MULTIBOOT int __gba_multiboot;
MULTIBOOT

//add support for the rand function
#include <stdlib.h>

//declare the function prototype
void DrawPixel4(int x, int y, unsigned char bColor);

//declare some defines for the video mode
#define MODE_4 0x4
#define BG2_ENABLE 0x400
#define REG_DISPCNT *(unsigned int*)0x4000000
#define RGB(r,g,b) (unsigned short)((r)+((g)<<5)+((b)<<10))

//create a pointer to the video and palette buffers
unsigned short* videoBuffer = (unsigned short*)0x6000000;
unsigned short* paletteMem = (unsigned short*)0x5000000;

//changes the video mode
#define SetMode(mode) REG_DISPCNT = (mode)

/////////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
/////////////////////////////////////////////////////////////
int main(void)
{
    int x1,y1,n;

    SetMode(MODE_4 | BG2_ENABLE);

    for (n = 1; n < 256; n++)
        paletteMem[n] = RGB(rand() % 31, rand() % 31, rand() % 31);

    while(1)
    {
        x1 = rand() % 240;
        y1 = rand() % 160;

        DrawPixel4(x1, y1, rand() % 256);
   }

    return 0;
}

/////////////////////////////////////////////////////////////
// Function: DrawPixel4
// Draws a pixel in mode 4
/////////////////////////////////////////////////////////////
void DrawPixel4(int x, int y, unsigned char color)
```

```
{
     unsigned short pixel;
    unsigned short offset = (y * 240 + x) >> 1;

     pixel = videoBuffer[offset];
     if (x & 1)
          videoBuffer[offset] = (color << 8) + (pixel & 0x00FF);
    else
         videoBuffer[offset] = (pixel & 0xFF00) + color;
}
```

The output from the Mode4Pixels program is shown in Figure 5.10.


PE Visual HAM - [G:\GBA\Sources\CHAPTE~3\MODE4P~1\main.c]
File Edit Format View Project Advanced External Tools Window Help
HAM Workspace
Source Files
main.c
Header Files
Files
Auto Complete
VisualBoyAdvance
File Options Cheats Tools Help
Ln 4, Col 8 SelLen: 0 INS

```
{
    int x1,y1,n;

    SetMode(MODE_4 | BG2_ENABLE);

    for (n = 1; n < 256; n++)
        paletteMem[n] = RGB(rand() % 31, rand() % 31, ran

    while(1)
    {
        x1 = rand() % 240;
        y1 = rand() % 160;

        DrawPixel4(x1, y1, rand() % 256);
    }

    return 0;
}

/////////////////////////////////////////////////////////////
// Function: DrawPixel4
// Draws a pixel in mode 4
/////////////////////////////////////////////////////////////
void DrawPixel4(int x, int y, unsigned char color)
{
    unsigned short pixel;
    unsigned short offset = (y * 240 + x) >> 1;

    pixel = videoBuffer[offset];
    if (x & 1)
        videoBuffer[offset] = (color << 8) + (pixel & 0x00FF);
    else
        videoBuffer[offset] = (pixel & 0xFF00) + color;
}
```

```
g:/ham/gcc-arm/bin/arm-thumb-elf-ld.exe    -L g:/ham/gcc-arm/lib/gcc-lib/arm-thumb-elf/3.2.2/normal -L g:/ham/gcc-arm/arm-thumb-elf/lib/normal -L g:/ham/gcc-arm,
g:/ham/gcc-arm/bin/arm-thumb-elf-objcopy.exe    -v -O binary Mode4Pixels.elf Mode4Pixels.gba
copy from Mode4Pixels.elf(elf32-littlearm) to Mode4Pixels.gba(binary)
g:/ham/tools/win32/rm -f *.o *.i *.ii
g:/ham/tools/win32/gbafix.exe   Mode4Pixels.gba
ROM fixed!
g:/ham/tools/win32/vba.exe   Mod
```

Figure 5.10

The Mode4Pixels program draws random pixels on the GBA screen.

Using this new DrawPixel4 function and the sample program, you should be able to modify the mode 3 samples for drawing lines, circles, and filled boxes. I won't waste space by listing programs when the only difference is in the DrawPixel function (DrawPixel3, DrawPixel4, and DrawPixel5).. But I encourage you to plug DrawPixel4 into those projects to see how well it works.

Drawing Bitmaps

Since mode 4 is an 8-bit mode (as well you know at this point), you can't use the same file generated by gfx2gba for mode 4 that you used for mode 3. For one thing, there are only 38,400 bytes in a mode 4 image (that is, a full-screen background image), while mode 3 backgrounds are 76,800 bytes (exactly twice the size). Let me show you how to convert a bitmap that is similar to the mode3.bmp file. This time, I'll call the new file mode4.bmp. Just use your favorite graphics editing program to create a new 240 x 160 image, and make sure it's 8-bit, with 256 colors, the only format supported by gfx2gba. If you prefer, you may look at the project folder on the CD-ROM under \Sources\Chapter05\Mode4Bitmap. Here's the command to convert the bitmap to a C array:

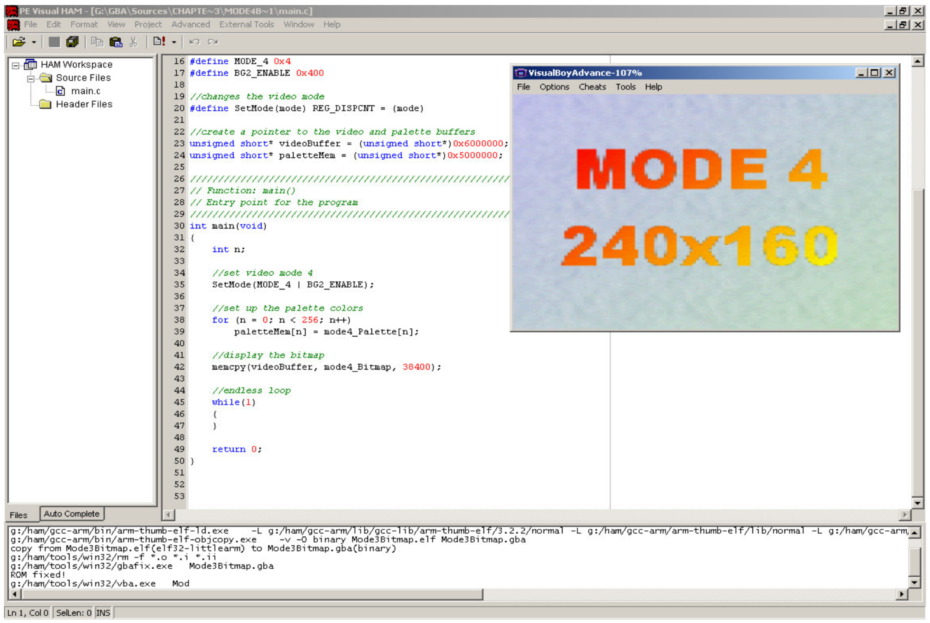
```
gfx2gba -fsrc -pmode4.pal mode4.bmp
```

These parameters for gfx2gba generate two files: mode4.raw.c and mode4.pal.c, containing the bitmap and palette, respectively. Setting up the palette for a bitmap is similar to setting it up for drawing pixels, except that now you use the mode4_Palette array instead of random numbers. I've written another program to show how to display a bitmap in mode 4. This project is called Mode4Bitmap and is similar to the Mode3Bitmap program. Just create a new project in Visual HAM and replace all the default code in main.c with this code and run it by pressing F7.

```
//////////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 5: Bitmap-Based Video Modes
// Mode4Bitmap Project
// main.c source code file
//////////////////////////////////////////////////////////////

#define MULTIBOOT int __gba_multiboot;
MULTIBOOT

//include files
#include <stdlib.h>
#include <string.h>
#include "mode4.raw.c"
#include "mode4.pal.c"

//declare some defines for the video mode
#define REG_DISPCNT *(unsigned long*)0x4000000
#define MODE_4 0x4
```

```
#define BG2_ENABLE 0x400

//changes the video mode
#define SetMode(mode) REG_DISPCNT = (mode)

//create a pointer to the video and palette buffers
unsigned short* videoBuffer = (unsigned short*)0x6000000;
unsigned short* paletteMem = (unsigned short*)0x5000000;

/////////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
/////////////////////////////////////////////////////////////
int main(void)
{
    int n;

    //set video mode 4
    SetMode(MODE_4 | BG2_ENABLE);

    //set up the palette colors
    for (n = 0; n < 256; n++)
        paletteMem[n] = mode4_Palette[n];

    //display the bitmap
    memcpy(videoBuffer, mode4_Bitmap, 38400);

    //endless loop
    while(1)
    {
    }

    return 0;
}
```

Whoa, wait a second! This program isn't like Mode3Bitmap at all. Where's the DrawPixel4 function? I'm sure you noticed that memcpy function (which was the reason this program needed to include string.h). The memcpy function copies a specified number of bytes from a source buffer to a destination buffer. It works great for displaying a bitmap in mode 4! Figure 5.11 shows the output from the program. This ANSI C function works in exactly the same manner as it does on other systems. In fact, the same can be said about all of the C libraries included with the HAM SDK, as all are ANSI C compliant and exactly the same functions that you will find on any platform. This is good news for cross-platform development!


PE Visual HAM - [G:\GBA\Sources\CHAPTE~3\MODE4B~1\main.c]
File Edit Format View Project Advanced External Tools Window Help
HAM Workspace
Source Files
main.c
Header Files
Files
Auto Complete
VisualBoyAdvance-107%
File Options Cheats Tools Help
MODE 4
240x160
Ln 1, Col 0
SelLen: 0
INS

```
#define MODE_4 0x4
#define BG2_ENABLE 0x400

//changes the video mode
#define SetMode(mode) REG_DISPCNT = (mode)

//create a pointer to the video and palette buffers
unsigned short* videoBuffer = (unsigned short*)0x6000000;
unsigned short* paletteMem = (unsigned short*)0x5000000;

//////////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
//////////////////////////////////////////////////////////////
int main(void)
{
    int n;

    //set video mode 4
    SetMode(MODE_4 | BG2_ENABLE);

    //set up the palette colors
    for (n = 0; n < 256; n++)
        paletteMem[n] = mode4_Palette[n];

    //display the bitmap
    memcpy(videoBuffer, mode4_Bitmap, 38400);

    //endless loop
    while(1)
    {
    }

    return 0;
}
```

```
g:/ham/gcc-arm/bin/arm-thumb-elf-ld.exe    -L g:/ham/gcc-arm/lib/gcc-lib/arm-thumb-elf/3.2.2/normal -L g:/ham/gcc-arm/arm-thumb-elf/lib/normal -L g:/ham/gcc-arm
g:/ham/gcc-arm/bin/arm-thumb-elf-objcopy.exe    -v -O binary Mode3Bitmap.elf Mode3Bitmap.gba
copy from Mode3Bitmap.elf(elf32-littlearm) to Mode3Bitmap.gba(binary)
g:/ham/tools/win32/rm -f *.o *.i *.ii
g:/ham/tools/win32/gbafix.exe   Mode3Bitmap.gba
ROM fixed!
g:/ham/tools/win32/vba.exe   Mod
```

Figure 5.11
The Mode4Bitmap program shows how to display a bitmap in mode 4 using memcpy.

There are other ways to display a bitmap too. Memcpy may not be the fastest method, because it doesn't take advantage of writing two pixels at a time—it just copies one whole buffer to the screen. However, it might be faster to copy mode4_Bitmap to the videoBuffer using a loop that iterates 120 x 160 times (that's half the number of pixels). You may try this if you wish, but I'm not going to get into optimization at this point because DMA is faster than any software loop.

Page Flipping and Double Buffering

Page flipping is built into modes 4 and 5, so I'll show you how to use this great feature. Basically, when you draw everything into an off-screen buffer, things move along much more quickly. In fact, using a double buffer makes the drawing operations so fast that it interferes with the vertical refresh, so you must add code to check for the vertical blank period and only flip the page during this period.

In order to do page flipping, the program needs two video buffers instead of one—the front buffer and the rear buffer. These take up the same amount of video memory as the mode 3 buffer, and like I said, this is all part of the architecture of mode 4. The front buffer is located at 0 x 6000000 like usual, while the back buffer is located at 0x600A000. Here are the definitions:

```
unsigned short* FrontBuffer = (unsigned short*)0x6000000;

unsigned short* BackBuffer = (unsigned short*)0x600A000;
```

Using a back buffer bit modifier:

```
#define BACKBUFFER 0x10
```

along with the video mode register, we can write a function to flip from the front to the back buffer by simply changing the video buffer pointer. This simply redirects the GBA from one buffer to the other automatically—without requiring you to copy pixels!

```
void FlipPage(void)

{

     if(REG_DISPCNT & BACKBUFFER)

     {

          REG_DISPCNT &= ~BACKBUFFER;

          videoBuffer = BackBuffer;

     }

     else

     {

          REG_DISPCNT |= BACKBUFFER;

          videoBuffer = FrontBuffer;

     }

}
```

Here is the source code listing for the Mode4Flip program. You may open this project directly off the CD-ROM, located in \Sources\Chapter05\Mode4Flip, although all of these programs are short enough to be simply typed into Visual HAM and run directly. The exception arises when using media resources (bitmaps and waves), at which point you may want to copy the converted media files if you are still not fluent in the process of converting files to the GBA yet.

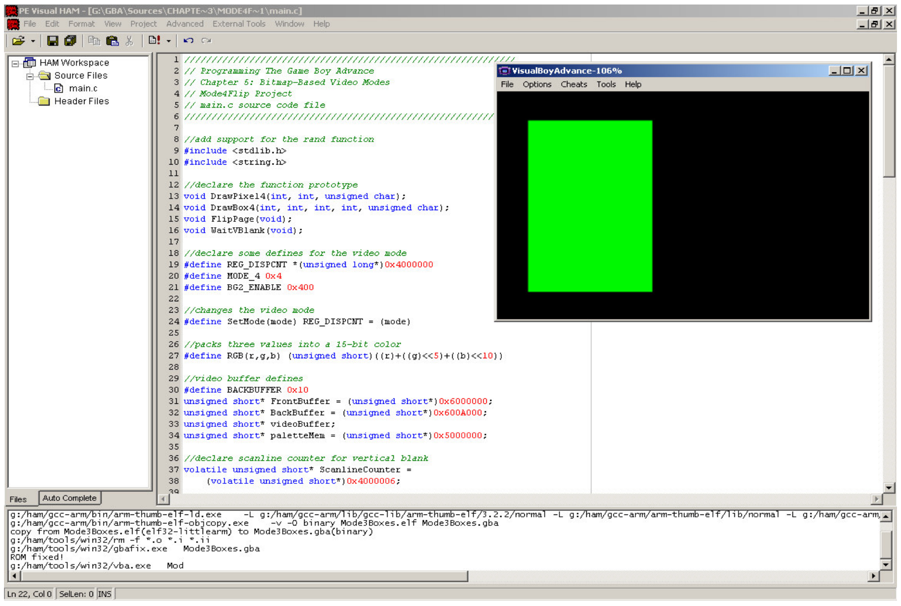
```
//////////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 5: Bitmap-Based Video Modes
// Mode4Flip Project
// main.c source code file
//////////////////////////////////////////////////////////////

#define MULTIBOOT int __gba_multiboot;
MULTIBOOT

//add support for the rand function
#include <stdlib.h>
#include <string.h>

//declare the function prototype
void DrawPixel4(int, int, unsigned char);
void DrawBox4(int, int, int, int, unsigned char);
void FlipPage(void);
void WaitVBlank(void);

//declare some defines for the video mode
#define REG_DISPCNT *(unsigned long*)0x4000000
#define MODE_4 0x4
#define BG2_ENABLE 0x400

//changes the video mode
#define SetMode(mode) REG_DISPCNT = (mode)

//packs three values into a 15-bit color
#define RGB(r,g,b) (unsigned short)((r)+((g)<<5)+((b)<<10))

//video buffer defines
```

```
#define BACKBUFFER 0x10
unsigned short* FrontBuffer = (unsigned short*)0x6000000;
unsigned short* BackBuffer = (unsigned short*)0x600A000;
unsigned short* videoBuffer;
unsigned short* paletteMem = (unsigned short*)0x5000000;

volatile unsigned short* ScanlineCounter =
    (volatile unsigned short*)0x4000006;

/////////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
/////////////////////////////////////////////////////////////
int main(void)
{
    int n;

    //set video mode and start page flipping
    SetMode(MODE_4 | BG2_ENABLE);
    FlipPage();

    //set the first two palette entries
    paletteMem = RGB(0, 31, 0);
    paletteMem = RGB(31, 0, 0);

    //draw the first random box
     DrawBox4(20, 20, 100, 140, 1);

    //flip the page to the back buffer
    FlipPage();

    //draw the second random box
     DrawBox4(140, 20, 220, 140, 2);
```

```
        while(1)
        {
            //wait for vertical blank
            WaitVBlank();

            //flip the page
            FlipPage();

            //slow it down--modify as needed
            n = 500000;
            while(n--);
       }

       return 0;
    }

    //////////////////////////////////////////////////////////////
    // Function: DrawPixel4
    // Draws a pixel in mode 4
    //////////////////////////////////////////////////////////////
    void DrawPixel4(int x, int y, unsigned char color)
    {
         unsigned short pixel;
        unsigned short offset = (y * 240 + x) >> 1;

         pixel = videoBuffer[offset];
         if (x & 1)
             videoBuffer[offset] = (color << 8) + (pixel & 0x00FF);
        else
            videoBuffer[offset] = (pixel & 0xFF00) + color;
    }


    //////////////////////////////////////////////////////////////
```

```
// Function: DrawBox4
// Draws a filled box
//////////////////////////////////////////////////////////////
void DrawBox4(int left, int top, int right, int bottom,
    unsigned char color)
{
     int x, y;

    for(y = top; y < bottom; y++)
        for(x = left; x < right; x++)
            DrawPixel4(x, y, color);
}

//////////////////////////////////////////////////////////////
// Function: FlipPage
// Switches between the front and back buffers
//////////////////////////////////////////////////////////////
void FlipPage(void)
{
     if(REG_DISPCNT & BACKBUFFER)
     {
          REG_DISPCNT &= ~BACKBUFFER;
          videoBuffer = BackBuffer;
     }
     else
     {
          REG_DISPCNT |= BACKBUFFER;
          videoBuffer = FrontBuffer;
     }
}
//////////////////////////////////////////////////////////////
// Function: WaitVBlank
// Checks the scanline counter for the vertical blank period
/////////////////////////////////////////////////////////////
void WaitVBlank(void)
{
     while(*ScanlineCounter < 160);

}
```

The output from the Mode4Flip program is shown in Figure 5.12.


PE Visual HAM - [G:\GBA\Sources\CHAPTE~3\MODE4F~1\main.c]
File Edit Format View Project Advanced External Tools Window Help
HAM Workspace
Source Files
main.c
Header Files
Files
Auto Complete
VisualBoyAdvance-106%
File Options Cheats Tools Help
Ln 22, Col 0
SelLen: 0
INS

```
/////////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 5: Bitmap-Based Video Modes
// Mode4Flip Project
// main.c source code file
/////////////////////////////////////////////////////////////

//add support for the rand function
#include <stdlib.h>
#include <string.h>

//declare the function prototype
void DrawPixel4(int, int, unsigned char);
void DrawBox4(int, int, int, int, unsigned char);
void FlipPage(void);
void WaitVBlank(void);

//declare some defines for the video mode
#define REG_DISPCNT *(unsigned long*)0x4000000
#define MODE_4 0x4
#define BG2_ENABLE 0x400

//changes the video mode
#define SetMode(mode) REG_DISPCNT = (mode)

//packs three values into a 15-bit color
#define RGB(r,g,b) (unsigned short)((r)+((g)<<5)+((b)<<10))

//video buffer defines
#define BACKBUFFER 0x10
unsigned short* FrontBuffer = (unsigned short*)0x6000000;
unsigned short* BackBuffer = (unsigned short*)0x600A000;
unsigned short* videoBuffer;
unsigned short* paletteMem = (unsigned short*)0x5000000;

//declare scanline counter for vertical blank
volatile unsigned short* ScanlineCounter =
    (volatile unsigned short*)0x4000006;
```

```
g:/ham/gcc-arm/bin/arm-thumb-elf-ld.exe    -L g:/ham/gcc-arm/lib/gcc-lib/arm-thumb-elf/3.2.2/normal -L g:/ham/gcc-arm/arm-thumb-elf/lib/normal -L g:/ham/gcc-arm,
g:/ham/gcc-arm/bin/arm-thumb-elf-objcopy.exe    -v -O binary Mode3Boxes.elf Mode3Boxes.gba
copy from Mode3Boxes.elf(elf32-littlearm) to Mode3Boxes.gba(binary)
g:/ham/tools/win32/rm -f *.o *.i *.ii
g:/ham/tools/win32/gbafix.exe   Mode3Boxes.gba
ROM fixed!
g:/ham/tools/win32/vba.exe   Mod
```

Figure 5.12
The Mode4Flip program demonstrates how page flipping works.

Working with Mode 5

Mode 5 is similar to mode 3 in that it features 15-bit pixels and therefore has no need for a palette. However, mode 5 has a lower resolution than either of the previous two modes: only 160 x 128! That is much smaller than 240 x 160, but it's a compromise in that you get a double buffer and also 15-bit color.

Drawing Pixels

Mode 5 is so similar to mode 3 that it doesn't require a lengthy explanation. Basically, you can just modify the DrawPixel3 function so that it takes into account the more limited screen resolution of mode 5 and come up with a new function:

```
void DrawPixel5(int x, int y, unsigned short c)
```

```
{
    videoBuffer[y * 160 + x] = c;
}
```

Testing Mode 5

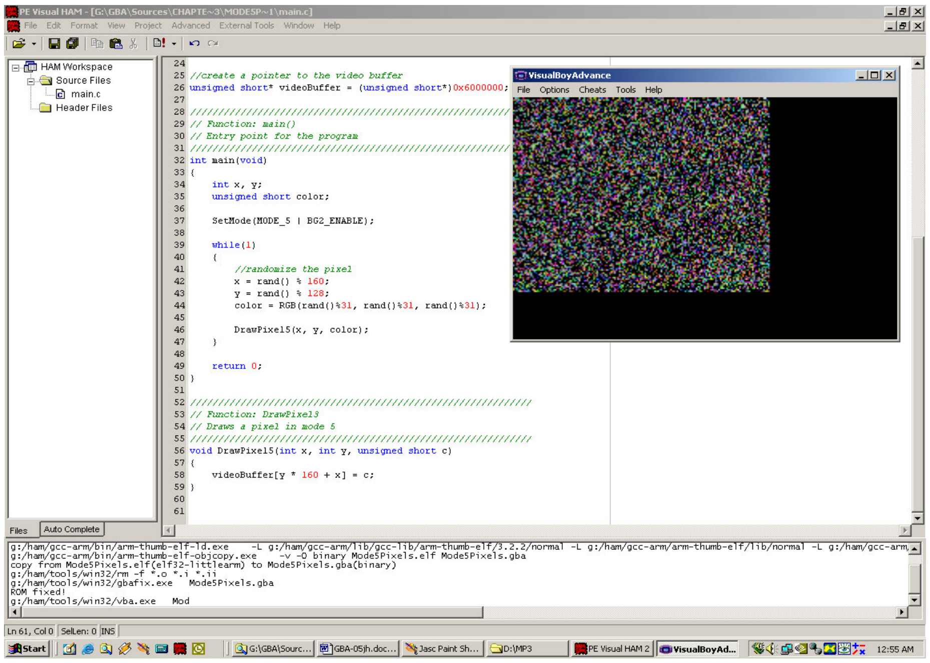
The Mode5Pixels program is similar to the pixel program for mode 3. Basically, just change the SetMode line so it uses MODE_5 and change the range of the random numbers to take into account the 160 x 128 resolution. Drawing bitmaps is precisely the same for mode 5 as it is for mode 3, the only exception being the smaller size. If you want to write a bitmap display program for mode 5, just create a bitmap image that is 160 x 128 pixels in size, then run gfx2gba using the same parameters for mode 3, and use mode5_Bitmap instead of mode3_Bitmap. Simple! Here is the complete Mode5Pixels program:

```
//////////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 5: Bitmap-Based Video Modes
// Mode5Pixels Project
// main.c source code file
//////////////////////////////////////////////////////////////

//add support for the rand function
#include <stdlib.h>

//declare the function prototype
void DrawPixel5(int, int, unsigned short);

//declare some defines for the video mode
#define REG_DISPCNT *(unsigned long*)0x4000000
#define MODE_5 0x5
#define BG2_ENABLE 0x400

//changes the video mode
#define SetMode(mode) REG_DISPCNT = (mode)
```

```
//packs three values into a 15-bit color
#define RGB(r,g,b) ((r)+(g<<5)+(b<<10))

//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;

/////////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
/////////////////////////////////////////////////////////////
int main(void)
{
    int x, y;
    unsigned short color;

    SetMode(MODE_5 | BG2_ENABLE);

    while(1)
    {
        //randomize the pixel
    x = rand() % 160;
        y = rand() % 128;
        color = RGB(rand()%31, rand()%31, rand()%31);
        DrawPixel5(x, y, color);
    }

    return 0;
}

/////////////////////////////////////////////////////////////
// Function: DrawPixel5
// Draws a pixel in mode 5
/////////////////////////////////////////////////////////////
void DrawPixel5(int x, int y, unsigned short c)
```

```
{
   videoBuffer[y * 160 + x] = c;
}
```

The output from the Mode5Pixels program is shown in Figure 5.13.


PE Visual HAM - [G:\GBA\Sources\CHAPTE~3\MODE5P~1\main.c]
File Edit Format View Project Advanced External Tools Window Help
HAM Workspace
Source Files
main.c
Header Files
Files
Auto Complete
VisualBoyAdvance
File Options Cheats Tools Help
Ln 61, Col 0 SelLen: 0 INS
Start
G:\GBA\Sourc...
GBA-05jh.doc...
Jasc Paint Sh...
D:\MP3
PE Visual HAM 2
VisualBoyAd...
12:55 AM

```
//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;

/////////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
/////////////////////////////////////////////////////////////
int main(void)
{
    int x, y;
    unsigned short color;

    SetMode(MODE_5 | BG2_ENABLE);

    while(1)
    {
        //randomize the pixel
        x = rand() % 160;
        y = rand() % 128;
        color = RGB(rand()%31, rand()%31, rand()%31);

        DrawPixel5(x, y, color);
    }

    return 0;
}

/////////////////////////////////////////////////////////////
// Function: DrawPixel3
// Draws a pixel in mode 5
/////////////////////////////////////////////////////////////
void DrawPixel5(int x, int y, unsigned short c)
{
    videoBuffer[y * 160 + x] = c;
}
```

```
g:/ham/gcc-arm/bin/arm-thumb-elf-ld.exe    -L g:/ham/gcc-arm/lib/gcc-lib/arm-thumb-elf/3.2.2/normal -L g:/ham/gcc-arm/arm-thumb-elf/lib/normal -L g:/ham/gcc-arm
g:/ham/gcc-arm/bin/arm-thumb-elf-objcopy.exe    -v -O binary Mode5Pixels.elf Mode5Pixels.gba
copy from Mode5Pixels.elf(elf32-littlearm) to Mode5Pixels.gba(binary)
g:/ham/tools/win32/rm -f *.o *.i *.ii
g:/ham/tools/win32/gbafix.exe   Mode5Pixels.gba
ROM fixed!
g:/ham/tools/win32/vba.exe   Mod
```

Figure 5.13

The Mode5Pixels program draws random pixels on the GBA screen.

Printing Text on the Screen

At this point I believe we've conquered the bitmapped video modes on the GBA, so I'd like to talk about a very important subject that is directly related to the subjects already covered, and that is text output. You will almost immediately find a need to display text on the screen, in order to present the player with a game menu, to display messages on the screen . . . whatever! Text output is not built into the GBA, although it is a feature available in Hamlib (as you saw back in Chapter 4, "Starting with the Basics" with the Greeting program).

Text output must be done the hard way, just like drawing lines, circles, and boxes; that is, you must write code to display the pixels of each character in a font that you must create from scratch. Now, I realize that many programmers use a bitmapped font, and that's not a bad idea at all, because the font characters can be treated as sprites. However, I prefer a low-memory footprint and more control over the font display mechanism. I have written two functions to display the font (which I will cover shortly). The Print function accepts a location, string, and color for the text output. The DrawChar function actually does the work of drawing each pixel in the font character, which is passed one at a time from the Print function.

```
void Print(int left, int top, char *str, unsigned short color)
{
    int pos = 0;
    while (*str)
    {
        DrawChar(left + pos, top, *str++, color);
        pos += 8;
    }
}

void DrawChar(int left, int top, char letter, unsigned short color)
{
    int x, y;
    int draw;

    for(y = 0; y < 8; y++)
        for (x = 0; x < 8; x++)
        {
            // grab a pixel from the font char
            draw = font[(letter-32) * 64 + y * 8 + x];
            // if pixel = 1, then draw it
            if (draw)
                DrawPixel3(left + x, top + y, color);
        }
}
```

The Hard-Coded Font

Now, to make sense of these text output functions, you'll first need a font. I have created a font for this purpose; it is just a large array of numbers. To make the font source easier to read, I defined the letter *W* to represent 1, which really helps when typing in the code. Of course, you may grab this font.h file from the CD-ROM. It is located in

\Sources\Chapter05\DrawText. If you are typing in the code, you'll want to create a new project in Visual HAM called DrawText and add a new file to the project called font.h, which you may type the following code into. This font file will be used by nearly every sample program from this point forward.

```
#ifndef _FONT_H
#define _FONT_H

#define W 1

unsigned short font[] =
{
// (space) 32
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
// ! 33
    0,0,W,W,0,0,0,0,
    0,0,W,W,0,0,0,0,
    0,0,W,W,0,0,0,0,
    0,0,W,W,0,0,0,0,
    0,0,W,W,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,W,W,0,0,0,0,
    0,0,W,W,0,0,0,0,
// " 34
    0,0,0,0,0,0,0,0,
    0,W,W,0,W,W,0,0,
    0,W,W,0,W,W,0,0,
    0,0,W,0,0,W,0,0,
    0,0,W,0,0,W,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
// # 35
    0,0,0,0,0,0,0,0,
    0,0,W,0,0,W,0,0,
    0,W,W,W,W,W,W,0,
    0,0,W,0,0,W,0,0,
    0,0,W,0,0,W,0,0,
    0,W,W,W,W,W,W,0,
    0,0,W,0,0,W,0,0,
    0,0,0,0,0,0,0,0,
// $ 36
    0,0,0,W,0,0,0,0,
    0,0,W,W,W,W,0,0,
    0,W,0,W,0,0,0,0,
    0,0,W,W,W,0,0,0,
    0,0,0,W,0,W,0,0,
    0,W,W,W,W,0,0,0,
    0,0,0,W,0,0,0,0,
    0,0,0,0,0,0,0,0,
// % 37
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,W,W,0,0,W,0,0,
    0,W,W,0,W,0,0,0,
    0,0,0,W,0,0,0,0,
    0,0,W,0,W,W,0,0,
    0,W,0,0,W,W,0,0,
    0,0,0,0,0,0,0,0,
// & 38
    0,0,0,0,0,0,0,0,
    0,0,W,W,W,0,0,0,
    0,W,0,0,0,W,0,0,
    0,0,W,W,W,0,0,0,
    0,W,0,W,0,0,0,0,
    0,W,0,0,W,0,W,0,
    0,0,W,W,W,W,0,0,
    0,0,0,0,0,0,W,0,
// ' 39
    0,0,0,W,W,0,0,0,
    0,0,0,W,W,0,0,0,
    0,0,0,0,W,0,0,0,
    0,0,0,0,W,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
// ( 40
    0,0,0,0,0,0,W,0,
    0,0,0,0,0,W,0,0,
    0,0,0,0,0,W,0,0,
    0,0,0,0,0,W,0,0,
    0,0,0,0,0,W,0,0,
    0,0,0,0,0,W,0,0,
    0,0,0,0,0,W,0,0,
    0,0,0,0,0,0,W,0,
// ) 41
    0,W,0,0,0,0,0,0,
    0,0,W,0,0,0,0,0,
    0,0,W,0,0,0,0,0,
    0,0,W,0,0,0,0,0,
    0,0,W,0,0,0,0,0,
    0,0,W,0,0,0,0,0,
    0,0,W,0,0,0,0,0,
    0,W,0,0,0,0,0,0,
// * 42
```

```
    0,0,0,0,0,0,0,0,
    0,0,0,W,0,0,0,0,
    0,W,0,W,0,W,0,0,
    0,0,W,W,W,0,0,0,
    0,0,W,W,W,0,0,0,
    0,0,W,W,W,0,0,0,
    0,W,0,0,0,W,0,0,
    0,0,0,0,0,0,0,0,
// + 43
    0,0,0,0,0,0,0,0,
    0,0,0,W,W,0,0,0,
    0,0,0,W,W,0,0,0,
    0,W,W,W,W,W,W,0,
    0,W,W,W,W,W,W,0,
    0,0,0,W,W,0,0,0,
    0,0,0,W,W,0,0,0,
    0,0,0,0,0,0,0,0,
// , 44
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,W,0,0,0,0,
    0,0,W,W,0,0,0,0,
    0,0,W,0,0,0,0,0,
// - 45
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,W,W,W,W,W,W,0,
    0,W,W,W,W,W,W,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
// . 46
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,W,W,0,0,0,0,
    0,0,W,W,0,0,0,0,
    0,0,0,0,0,0,0,0,
// / 47
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,W,0,
    0,0,0,0,0,W,W,0,
    0,0,0,0,W,W,0,0,
    0,0,0,W,W,0,0,0,
    0,0,W,W,0,0,0,0,
    0,W,W,0,0,0,0,0,
    0,W,0,0,0,0,0,0,
// 0 48
    0,0,W,W,W,W,0,0,
    0,W,W,0,0,0,W,0,
    0,W,W,0,0,W,W,0,
    0,W,W,0,W,0,W,0,
    0,W,W,W,0,0,W,0,
    0,W,W,0,0,0,W,0,
    0,W,W,0,0,0,W,0,
    0,0,W,W,W,W,0,0,
// 1 49
    0,0,0,0,W,0,0,0,
    0,0,W,W,W,0,0,0,
    0,0,0,W,W,0,0,0,
    0,0,0,W,W,0,0,0,
    0,0,0,W,W,0,0,0,
    0,0,0,W,W,0,0,0,
    0,0,0,W,W,0,0,0,
    0,0,W,W,W,W,0,0,
// 2 50
    0,0,W,W,W,W,0,0,
    0,W,W,W,W,W,W,0,
    0,W,0,0,0,0,W,0,
    0,0,0,0,0,W,W,0,
    0,0,0,0,W,W,0,0,
    0,0,0,W,W,0,0,0,
    0,0,W,W,0,0,0,0,
    0,W,W,W,W,W,W,0,
// 3 51
    0,0,W,W,W,W,0,0,
    0,W,W,W,W,W,W,0,
    0,W,0,0,0,0,W,0,
    0,0,0,0,0,0,W,0,
    0,0,0,W,W,W,0,0,
    0,0,0,0,0,0,W,0,
    0,W,0,0,0,0,W,0,
    0,0,W,W,W,W,0,0,
// 4 52
    0,0,0,0,W,W,0,0,
    0,0,0,W,W,W,0,0,
    0,0,W,W,0,W,0,0,
    0,W,W,0,0,W,0,0,
    0,W,W,W,W,W,0,0,
    0,0,0,0,W,W,0,0,
    0,0,0,0,W,W,0,0,
    0,0,0,0,W,W,0,0,
// 5 53
    0,W,W,W,W,W,W,0,
    0,W,W,W,W,W,W,0,
    0,W,W,0,0,0,0,0,
    0,W,W,0,0,0,0,0,
    0,0,W,W,W,W,0,0,
    0,0,0,0,0,0,W,0,
    0,W,0,0,0,0,W,0,
    0,0,W,W,W,W,0,0,
// 6 54
    0,0,0,W,W,W,W,0,
    0,0,W,W,W,W,W,0,
    0,W,W,0,0,0,0,0,
    0,W,W,0,0,0,0,0,
    0,W,W,W,W,W,0,0,
    0,W,W,0,0,0,W,0,
    0,W,W,0,0,0,W,0,
    0,0,W,W,W,W,0,0,
// 7 55
    0,W,W,W,W,W,W,0,
    0,W,W,W,W,W,W,0,
    0,W,0,0,0,W,W,0,
    0,0,0,0,W,W,0,0,
    0,0,0,0,W,W,0,0,
    0,0,0,W,W,0,0,0,
    0,0,0,W,W,0,0,0,
    0,0,W,W,0,0,0,0,
// 8 56
    0,0,W,W,W,W,0,0,
    0,W,W,W,W,W,W,0,
    0,W,W,0,0,0,W,0,
    0,0,W,0,0,0,W,0,
    0,0,W,W,W,W,0,0,
    0,W,W,0,0,0,W,0,
    0,W,W,0,0,0,W,0,
    0,0,W,W,W,W,0,0,
// 9 57
    0,0,W,W,W,W,0,0,
    0,W,W,W,W,W,W,0,
    0,W,W,0,0,0,W,0,
    0,W,W,0,0,0,W,0,
    0,0,W,W,W,W,W,0,
    0,0,0,0,0,W,W,0,
    0,0,0,0,0,W,W,0,
    0,0,0,0,0,W,W,0,
// : 58
    0,0,0,0,0,0,0,0,
    0,0,W,W,0,0,0,0,
    0,0,W,W,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,W,W,0,0,0,0,
    0,0,W,W,0,0,0,0,
    0,0,0,0,0,0,0,0,
// ; 59
    0,0,W,W,0,0,0,0,
    0,0,W,W,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,W,W,0,0,0,0,
    0,0,W,W,0,0,0,0,
    0,W,W,0,0,0,0,0,
    0,W,W,0,0,0,0,0,
// < 60
    0,0,0,0,W,W,0,0,
    0,0,0,W,W,0,0,0,
    0,0,W,W,0,0,0,0,
    0,W,W,0,0,0,0,0,
    0,W,W,0,0,0,0,0,
    0,0,W,W,0,0,0,0,
    0,0,0,W,W,0,0,0,
    0,0,0,0,W,W,0,0,
// = 61
    0,0,0,0,0,0,0,0,
    0,W,W,W,W,W,0,0,
    0,W,W,W,W,W,0,0,
    0,0,0,0,0,0,0,0,
    0,W,W,W,W,W,0,0,
    0,W,W,W,W,W,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
// > 62
    0,W,W,0,0,0,0,0,
    0,0,W,W,0,0,0,0,
    0,0,0,W,W,0,0,0,
    0,0,0,0,W,W,0,0,
    0,0,0,0,W,W,0,0,
    0,0,0,W,W,0,0,0,
    0,0,W,W,0,0,0,0,
    0,W,W,0,0,0,0,0,
// ? 63
    0,0,W,W,W,W,0,0,
    0,W,W,W,W,W,W,0,
    0,W,0,0,0,W,W,0,
    0,0,0,0,W,W,0,0,
    0,0,0,W,W,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,W,W,0,0,0,
    0,0,0,W,W,0,0,0,
// @ 64
    0,0,0,0,0,0,0,0,
    0,0,W,W,W,0,0,0,
    0,W,0,0,0,W,0,0,
    0,W,0,W,W,0,0,0,
    0,W,0,W,W,0,0,0,
    0,W,0,0,0,0,W,0,
    0,0,W,W,W,W,0,0,
    0,0,0,0,0,0,0,0,
// A 65
    0,0,W,W,W,0,0,0,
    0,W,W,W,W,W,0,0,
    0,W,W,0,0,W,0,0,
    0,W,W,0,0,W,0,0,
    0,W,W,W,W,W,0,0,
    0,W,W,W,W,W,0,0,
    0,W,W,0,0,W,0,0,
    0,W,W,0,0,W,0,0,
// B 66
    0,W,W,W,W,W,0,0,
    0,W,W,W,W,W,W,0,
    0,W,W,0,0,W,W,0,
    0,W,W,0,0,W,0,0,
    0,W,W,W,W,W,0,0,
    0,W,W,0,0,W,W,0,
    0,W,W,0,0,W,W,0,
    0,W,W,W,W,W,0,0,
// C 67
    0,0,W,W,W,W,0,0,
    0,W,W,W,W,W,W,0,
    0,W,W,0,0,0,W,0,
    0,W,W,0,0,0,0,0,
    0,W,W,0,0,0,0,0,
    0,W,W,0,0,0,0,0,
    0,W,W,0,0,0,W,0,
    0,0,W,W,W,W,0,0,
// D 68
    0,W,W,W,W,W,0,0,
    0,W,W,W,W,W,W,0,
    0,W,W,0,0,0,W,0,
    0,W,W,0,0,0,W,0,
    0,W,W,0,0,0,W,0,
    0,W,W,0,0,0,W,0,
    0,W,W,0,0,0,W,0,
    0,W,W,W,W,W,0,0,
// E 69
    0,W,W,W,W,W,W,0,
    0,W,W,W,W,W,W,0,
    0,W,W,0,0,0,0,0,
    0,W,W,0,0,0,0,0,
    0,W,W,W,W,0,0,0,
    0,W,W,0,0,0,0,0,
    0,W,W,0,0,0,0,0,
    0,W,W,W,W,W,W,0,
// F 70
    0,W,W,W,W,W,W,0,
    0,W,W,W,W,W,W,0,
    0,W,W,0,0,0,0,0,
    0,W,W,0,0,0,0,0,
    0,W,W,W,W,0,0,0,
    0,W,W,0,0,0,0,0,
    0,W,W,0,0,0,0,0,
    0,W,W,0,0,0,0,0,
// G 71
    0,0,W,W,W,W,0,0,
    0,W,W,W,W,W,W,0,
    0,W,W,0,0,0,W,0,
    0,W,W,0,0,0,0,0,
    0,W,W,0,W,W,W,0,
    0,W,W,0,0,0,W,0,
    0,W,W,0,0,0,W,0,
    0,0,W,W,W,W,0,0,
// H 72
    0,W,W,0,0,0,W,0,
    0,W,W,0,0,0,W,0,
    0,W,W,0,0,0,W,0,
    0,W,W,0,0,0,W,0,
    0,W,W,W,W,W,W,0,
    0,W,W,0,0,0,W,0,
    0,W,W,0,0,0,W,0,
    0,W,W,0,0,0,W,0,
// I 73
    0,0,W,W,W,W,0,0,
    0,0,W,W,W,W,0,0,
    0,0,0,W,W,0,0,0,
    0,0,0,W,W,0,0,0,
    0,0,0,W,W,0,0,0,
    0,0,0,W,W,0,0,0,
    0,0,0,W,W,0,0,0,
    0,0,W,W,W,W,0,0,
// J 74
    0,0,0,0,W,W,0,0,
    0,0,0,0,W,W,0,0,
    0,0,0,0,W,W,0,0,
    0,0,0,0,W,W,0,0,
    0,0,0,0,W,W,0,0,
    0,W,0,0,W,W,0,0,
    0,W,W,W,W,W,0,0,
    0,0,W,W,W,0,0,0,
// K 75
    0,W,W,0,0,0,W,0,
    0,W,W,0,0,W,W,0,
    0,W,W,0,W,W,0,0,
    0,W,W,W,W,0,0,0,
    0,W,W,W,W,0,0,0,
    0,W,W,0,W,W,0,0,
    0,W,W,0,0,W,W,0,
    0,W,W,0,0,0,W,0,
// L 76
```

```
        0,W,W,0,0,0,0,0,
        0,W,W,0,0,0,0,0,
        0,W,W,0,0,0,0,0,
        0,W,W,0,0,0,0,0,
        0,W,W,0,0,0,0,0,
        0,W,W,0,0,0,0,0,
        0,W,W,0,0,0,0,0,
        0,W,W,W,W,W,W,0,
    // M 77
        0,W,W,0,0,0,W,0,
        0,W,W,0,0,0,W,0,
        0,W,W,W,0,W,W,0,
        0,W,W,W,W,W,W,0,
        0,W,W,0,W,0,W,0,
        0,W,W,0,0,0,W,0,
        0,W,W,0,0,0,W,0,
        0,W,W,0,0,0,W,0,
    // N 78
        0,W,W,0,0,0,W,0,
        0,W,W,0,0,0,W,0,
        0,W,W,W,0,0,W,0,
        0,W,W,W,W,0,W,0,
        0,W,W,0,W,W,W,0,
        0,W,W,0,0,W,W,0,
        0,W,W,0,0,0,W,0,
        0,W,W,0,0,0,W,0,
    // O 79
        0,0,W,W,W,W,0,0,
        0,W,W,W,W,W,W,0,
        0,W,W,0,0,0,W,0,
        0,W,W,0,0,0,W,0,
        0,W,W,0,0,0,W,0,
        0,W,W,0,0,0,W,0,
        0,W,W,0,0,0,W,0,
        0,0,W,W,W,W,0,0,
    // P 80
        0,W,W,W,W,W,0,0,
        0,W,W,W,W,W,W,0,
        0,W,W,0,0,0,W,0,
        0,W,W,0,0,0,W,0,
        0,W,W,W,W,W,0,0,
        0,W,W,0,0,0,0,0,
        0,W,W,0,0,0,0,0,
        0,W,W,0,0,0,0,0,
    // Q 81
        0,0,W,W,W,W,0,0,
        0,W,W,W,W,W,W,0,
        0,W,W,0,0,0,W,0,
        0,W,W,0,0,0,W,0,
        0,W,W,0,0,0,W,0,
        0,W,W,0,0,W,W,0,
        0,W,W,0,0,0,W,0,
        0,0,W,W,W,W,0,W,
    // R 82
        0,W,W,W,W,W,0,0,
        0,W,W,W,W,W,W,0,
        0,W,W,0,0,0,W,0,
        0,W,W,0,0,0,W,0,
        0,W,W,0,0,W,0,0,
        0,W,W,W,W,0,0,0,
        0,W,W,0,W,W,0,0,
        0,W,W,0,0,W,W,0,
    // S 83
        0,0,W,W,W,W,0,0,
        0,W,W,W,W,W,W,0,
        0,W,W,0,0,0,W,0,
        0,W,W,0,0,0,0,0,
        0,0,W,W,W,W,0,0,
        0,0,0,0,0,0,W,0,
        0,W,0,0,0,0,W,0,
        0,0,W,W,W,W,0,0,
    // T 84
        0,W,W,W,W,W,W,0,
        0,W,W,W,W,W,W,0,
        0,0,0,W,W,0,0,0,
        0,0,0,W,W,0,0,0,
        0,0,0,W,W,0,0,0,
        0,0,0,W,W,0,0,0,
        0,0,0,W,W,0,0,0,
        0,0,0,W,W,0,0,0,
    // U 85
        0,W,W,0,0,0,W,0,
        0,W,W,0,0,0,W,0,
        0,W,W,0,0,0,W,0,
        0,W,W,0,0,0,W,0,
        0,W,W,0,0,0,W,0,
        0,W,W,0,0,0,W,0,
        0,W,W,0,0,0,W,0,
        0,0,W,W,W,W,0,0,
    // V 86
        0,W,W,0,0,0,W,0,
        0,W,W,0,0,0,W,0,
        0,W,W,0,0,0,W,0,
        0,W,W,0,0,0,W,0,
        0,0,W,W,0,W,0,0,
        0,0,W,W,0,W,0,0,
        0,0,W,W,0,W,0,0,
        0,0,0,W,W,0,0,0,
    // W 87
        0,W,W,0,0,0,0,W,
        0,W,W,0,0,0,0,W,
        0,W,W,0,0,0,0,W,
        0,W,W,0,0,0,0,W,
        0,W,W,0,W,W,0,W,
        0,0,W,W,0,0,W,0,
        0,0,W,W,0,0,W,0,
        0,0,W,W,0,0,W,0,
    // X 88
        0,W,W,0,0,0,W,0,
        0,W,W,0,0,0,W,0,
        0,0,W,W,0,W,0,0,
        0,0,0,W,W,0,0,0,
        0,0,0,W,W,0,0,0,
        0,0,W,W,0,W,0,0,
        0,W,W,0,0,0,W,0,
        0,W,W,0,0,0,W,0,
    // Y 89
        0,W,W,0,0,0,W,0,
        0,W,W,0,0,0,W,0,
        0,W,W,0,0,0,W,0,
        0,0,W,W,W,W,0,0,
        0,0,0,W,W,0,0,0,
        0,0,0,W,W,0,0,0,
        0,0,0,W,W,0,0,0,
        0,0,0,W,W,0,0,0,
    // Z 90
        0,W,W,W,W,W,W,0,
        0,W,W,W,W,W,W,0,
        0,0,0,0,0,W,W,0,
        0,0,0,0,W,W,0,0,
        0,0,0,W,W,0,0,0,
        0,0,W,W,0,0,0,0,
        0,W,W,0,0,0,0,0,
        0,W,W,W,W,W,W,0,
    // [ 91
        0,0,0,0,W,W,W,0,
        0,0,0,0,W,W,0,0,
        0,0,0,0,W,W,0,0,
        0,0,0,0,W,W,0,0,
        0,0,0,0,W,W,0,0,
        0,0,0,0,W,W,0,0,
        0,0,0,0,W,W,0,0,
        0,0,0,0,W,W,W,0,
    // \ 92
        0,W,W,0,0,0,0,0,
        0,W,W,0,0,0,0,0,
        0,0,W,W,0,0,0,0,
        0,0,W,W,0,0,0,0,
        0,0,0,W,W,0,0,0,
        0,0,0,W,W,0,0,0,
        0,0,0,0,W,W,0,0,
        0,0,0,0,W,W,0,0,
    // ] 93
        0,W,W,W,0,0,0,0,
        0,W,W,W,0,0,0,0,
        0,0,0,W,0,0,0,0,
        0,0,0,W,0,0,0,0,
        0,0,0,W,0,0,0,0,
        0,0,0,W,0,0,0,0,
        0,0,0,W,0,0,0,0,
        0,W,W,W,0,0,0,0,
    };
    #endif
```

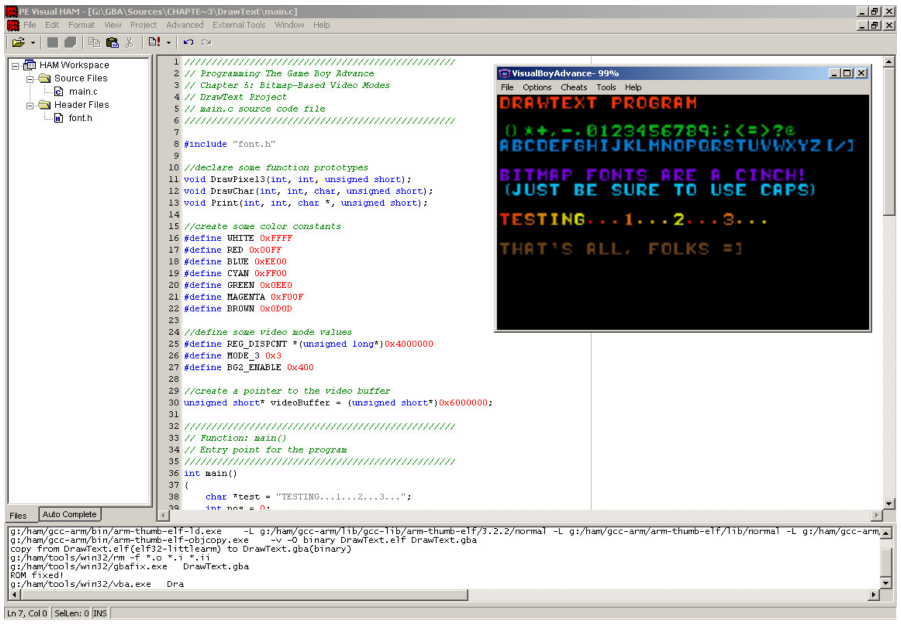
The DrawText Program

The DrawText program uses mode 3 to test the hard-coded font that you typed into the font.h file. Create a new project called DrawText and copy the font.h file into the folder for this new project. Following is the source code for the complete DrawText program that demonstrates the new font. If you would like to test this program using modes 4 or 5, you will need to change the DrawChar function so it calls DrawPixel4 or DrawPixel5, respectively, and then be sure to include those functions in the program.

```
//////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 5: Bitmap-Based Video Modes
// DrawText Project
// main.c source code file
//////////////////////////////////////////////////

#define MULTIBOOT int __gba_multiboot;
MULTIBOOT

#include "font.h"

//declare some function prototypes
void DrawPixel3(int, int, unsigned short);
void DrawChar(int, int, char, unsigned short);
void Print(int, int, char *, unsigned short);

//create some color constants
#define WHITE 0xFFFF
#define RED 0x00FF
#define BLUE 0xEE00
#define CYAN 0xFF00
#define GREEN 0x0EE0
#define MAGENTA 0xF00F
#define BROWN 0x0D0D

//define some video mode values
#define REG_DISPCNT *(unsigned long*)0x4000000
#define MODE_3 0x3
#define BG2_ENABLE 0x400

//create a pointer to the video buffer
```

```
unsigned short* videoBuffer = (unsigned short*)0x6000000;

/////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
/////////////////////////////////////////////////////
int main()
{
    char *test = "TESTING...1...2...3...";
    int pos = 0;

    //switch to video mode 3 (240x160 16-bit)
    REG_DISPCNT = (MODE_3 | BG2_ENABLE);

    Print(1, 1, "DRAWTEXT PROGRAM", RED);
    Print(1, 20, "()*+,-.0123456789:;<=>?@", GREEN);
    Print(1, 30, "ABCDEFGHIJKLMNOPQRSTUVWXYZ[/]", BLUE);
    Print(1, 50, "BITMAP FONTS ARE A CINCH!", MAGENTA);
    Print(1, 60, "(JUST BE SURE TO USE CAPS)", CYAN);

    //display each character in a different color
    while (*test)
    {
        DrawChar(1 + pos, 80, *test++, 0xBB + pos * 16);
        pos += 8;
    }

    Print(1, 100, "THAT'S ALL, FOLKS =]", BROWN);

    //continuous loop
    while(1)
    {
    }
    return 0;
}

//////////////////////////////////////////////////
// Function: DrawPixel3
// Draws a pixel in mode 3
//////////////////////////////////////////////////
void DrawPixel3(int x, int y, unsigned short color)
{
    videoBuffer[y * 240 + x] = color;
}

//////////////////////////////////////////////////
// Function: Print
// Prints a string using the hard-coded font
//////////////////////////////////////////////////
void Print(int left, int top, char *str, unsigned short color)
{
    int pos = 0;
    while (*str)
    {
        DrawChar(left + pos, top, *str++, color);
        pos += 8;
    }
}

//////////////////////////////////////////////////
// Function: DrawChar
// Draws a character one pixel at a time
//////////////////////////////////////////////////
void DrawChar(int left, int top, char letter, unsigned short color)
{
    int x, y;
    int draw;
    for(y = 0; y < 8; y++)
        for (x = 0; x < 8; x++)
        {
            // grab a pixel from the font char
        draw = font[(letter-32) * 64 + y * 8 + x];
        // if pixel = 1, then draw it
        if (draw)
                DrawPixel3(left + x, top + y, color);
        }
}
```

The output from the DrawText program is shown in Figure 5.14.


PE Visual HAM - [G:\GBA\Sources\CHAPTE~3\DrawText\main.c]
File Edit Format View Project Advanced External Tools Window Help
HAM Workspace
Source Files
main.c
Header Files
font.h
Files
Auto Complete
VisualBoyAdvance- 99%
File Options Cheats Tools Help
Ln 7, Col 0
SelLen: 0
INS

```
/////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 5: Bitmap-Based Video Modes
// DrawText Project
// main.c source code file
/////////////////////////////////////////////////////

#include "font.h"

//declare some function prototypes
void DrawPixel3(int, int, unsigned short);
void DrawChar(int, int, char, unsigned short);
void Print(int, int, char *, unsigned short);

//create some color constants
#define WHITE 0xFFFF
#define RED 0x00FF
#define BLUE 0xEE00
#define CYAN 0xFF00
#define GREEN 0x0EE0
#define MAGENTA 0xF00F
#define BROWN 0x0D0D

//define some video mode values
#define REG_DISPCNT *(unsigned long*)0x4000000
#define MODE_3 0x3
#define BG2_ENABLE 0x400

//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;

/////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
/////////////////////////////////////////////////////
int main()
{
    char *test = "TESTING...1...2...3...";
    int pos = 0;
```

```
DRAWTEXT PROGRAM

 () *+,-.0123456789:;<=>?@
ABCDEFGHIJKLMNOPQRSTUVWXYZ[/]

BITMAP FONTS ARE A CINCH!
 (JUST BE SURE TO USE CAPS)

TESTING...1...2...3...

THAT'S ALL, FOLKS =]
```

```
g:/ham/gcc-arm/bin/arm-thumb-elf-ld.exe    -L g:/ham/gcc-arm/lib/gcc-lib/arm-thumb-elf/3.2.2/normal -L g:/ham/gcc-arm/arm-thumb-elf/lib/normal -L g:/ham/gcc-arm
g:/ham/gcc-arm/bin/arm-thumb-elf-objcopy.exe    -v -O binary DrawText.elf DrawText.gba
copy from DrawText.elf(elf32-littlearm) to DrawText.gba(binary)
g:/ham/tools/win32/rm -f *.o *.i *.ii
g:/ham/tools/win32/gbafix.exe   DrawText.gba
ROM fixed!
g:/ham/tools/win32/vba.exe   Dra
```

Figure 5.14

The DrawText program uses the hard-coded font to print messages on the GBA screen.

Summary

This chapter conquered the extremely complex subject of doing graphics on the GBA using bitmapped video modes. Along the way, you learned about modes 3, 4, and 5 and how to draw pixels, lines, and filled boxes. This chapter also showed how to convert bitmap images to GBA format and then display them on the screen using either 8-bit mode 4 or 15-bit mode 3. Learning the ropes when it comes to graphics on a console is really the key to everything else in the games you are likely to write, because without a basic understanding of the graphics system, you will be unable to get even a simple Pong-style game on the screen. This chapter concluded by providing a useful font and functions for displaying text on the screen.

Challenges

The following challenges will help to reinforce the material you have learned in this chapter.

Challenge 1: Mode 5 was somewhat neglected in this chapter, but only because it is very similar to mode 3, and therefore I felt it would be repetitive to provide additional examples when only a single line of code is at stake. Now it's up to you! Write a program that displays a bitmap image using mode 5.

Challenge 2: The Mode3Boxes program could use some optimization. Now that you know how effective the memcpy and memset functions can be, this is a good opportunity to speed up the program. Modify the DrawBox3 function so it uses memcpy to fill a line of pixels on the screen, thus replacing the x loop. All you need to do is determine the number of x pixels across to use as the count for memset, and of course the color is used for the value parameter.

Challenge 3: Now for a real challenge! Add the font.h file and text functions to one of the mode 3 example programs, and then display the number of objects that have been drawn on the screen. If you want to add insult to injury, add the lowercase letters to the font so the Print function will be able to draw more of the ASCII character set.

Chapter Quiz

The following quiz will help to further reinforce the information covered in this chapter. The quiz consists of 10 multiple-choice questions with up to four possible answers. The key to the quiz may be found in the appendix.

1. What is the memory address for the video buffer?
   A. 0 x 7006000
   B. 0 x 6000000
   C. 0 x 6005000
   D. 0 x 4928300

2. Which video mode features a resolution of 240 x 160 and also a double buffer?
   A. Mode 4
   B. Mode 3
   C. Mode 5
   D. Mode 2

3. True or False: Video mode 5 uses a color palette.
   A. True
   B. False

4. Which video mode has a resolution of 160 x 128?
   A. Mode 6
   B. Mode 4
   C. Mode 5
   D. Mode 3

5. What is the address of the back buffer in mode 4?
   A. 0 x 6000000
   B. 0 x 7006000
   C. 0 x 6005000
   D. 0 x 600A000

6. What is the name of the line-drawing algorithm used in the Mode3Lines program?
   A. Einstein
   B. Bresenham
   C. Hawking
   D. Von Neumann

7. Which video mode has a resolution of 240 x 160, 15-bit color, and no back buffer?
   A. Mode 3
   B. Mode 2
   C. Mode 5
   D. Mode 4

8. What data type is used to reference the 15-bit video buffer for modes 3 and 5?
   A. unsigned char
   B. unsigned int

C. unsigned short
D. unsigned check

9. What is the color depth of the display in video mode 4?
A. 32 bits
B. 16 bits
C. 64 bits
D. 8 bits

10. What is the name of the organization that linked thousands of Game Boy Advance units together in order to create a supercomputer?
A. GBA-SETI
B. GBA White
C. Pocket Cray
D. I don't think so!