Chapter 4

Starting With The Basics

This chapter builds upon the previous one by continuing to increase your familiarity with the Visual HAM environment. It does so by developing several sample programs from scratch, showing you how to compile and run them. These programs are simple in nature and are not meant to focus on any particular subject (such as high-speed graphics, covered in later chapters). Instead, this chapter focuses on writing quick and simple programs using Hamlib and stock GBA code. If you are already familiar with GBA coding and you desire to get directly into graphics programming, you may skip this chapter and move on to the next one. The entire second part of the book, "Being One With The Pixel," is, as the name suggests, dedicated solely to graphics programming.

This chapter shows you how to write several complete programs from start to finish, so you will be prepared for the rest of the material in the book. The Visual HAM environment is capable of compiling any GBA program, but the HAM distribution comes with an excellent GBA wrapper library called Hamlib, which abstracts much of the low-level memory addressing code with actual function calls and callbacks that are just more intuitive and especially helpful for those who are new to console programming. The focus of this chapter is also to provide you with some experience writing, compiling, and running programs on either the emulator, multiboot, or flash linker and is therefore helpful for increasing your logistical skills with the development environment.

Specifically, this chapter covers the following subjects:

- The basics of a Game Boy program
- Displaying a friendly greeting
- Drawing pixels
- Filling the screen
- Detecting button presses
- Running programs directly on the GBA

The Basics Of A Game Boy Program

Programmers are impatient people. We want to see something happen as quickly as possible, even if it's not realistically humanly possible to do so. The motto of the programmer is often "Make it work, then fix it." Unfortunately, most of us love to write code but don't particularly like to design things. After all, it's far more fun to get started with hammer and nails rather than pencil and paper, right? This chapter is not about game design, although it is a subject that permeates the book, because design is part of the whole development process. The Game Boy Advance is a very simple computer, in the sense that it has a processor, has memory, and can run programs. The programs are meant for entertainment and don't always take the form of a game. There is a printer available for the Game Boy, for instance, and a digital camera that allows one to take photos and print them out. This is obviously not even remotely related to playing games, but it is fun nonetheless. Plus, you have to admit that it's cool being able to take pictures and print them out on a small handheld video game console!

Using the tools provided with this book and the knowledge gained from reading its contents, you have an opportunity to write any program you want for your own edification or entertainment. Many aftermarket accessories and programs are becoming available for GBA owners. For instance, there is an aftermarket MP3 player available for GBA! While I would debate the usefulness of such an accessory, there are many who would enjoy using their GBA to play music in addition to playing games. Given the versatility of the GBA platform, it's no wonder such things are commercially viable products. For example, one aftermarket accessory I purchased was the Afterburner backlight kit for my own personal GBA. I had a difficult time with the dark stock screen that is prone to light glare and simply could not deal with it while developing on it. Of course, at the present time, one can now just purchase a GBA SP with a built-in backlight and rechargeable batteries.

What Makes It Tick?

But I digress, the subject at hand is this: What makes a Game Boy Advance program tick? In a nutshell, a GBA program is just a binary ROM image that the hardware runs as if it were an integral part of the system. In other words, game cartridges fool the GBA into thinking the game was always there, because it isn't smart enough to know that you have removed the previous game and inserted a new one. But essentially, the GBA functions by assuming that a cartridge is a permanent piece of hardware. Let's dig into this line of thinking with more detail in the next section. From a programming perspective, how the GBA reads a cartridge is not as important at this point as knowing how to write GBA code in the first place, so I won't get into that extensively. You have already learned a great deal about the GBA hardware and how the console works from the last few chapters.

In many respects, these more difficult issues are what make Hamlib so appealing. When you consider that this library is a complete GBA framework capable of being used to produce commercial games, there is no reason to discount it as yet another wrapper library (a common complaint by software gurus). Hamlib is an awesome library that I will introduce to you in this chapter. But I recognize the valid point many programmers make in that they want to be as close to the hardware as possible—for many obvious reasons, such as knowing exactly what is going on, writing the fastest code possible, and so on. I share those sentiments, but I am also very encouraged by the strengths of Hamlib and the work put into the frequent updates to the library (as well as frequent updates to the HAM distribution in general).

So, how about if we abstract the hardware side of things from this point forward and focus on software development? If you are a programmer first (like I am) and a hardware hobbyist second, then the hardware is only of passing interest—which is most likely limited to knowing how a flash linker works, and so on. Now, on to what we programmers do best—writing source code.

A Friendly Greeting

How about a practical and easy-to-understand sample program to get things rollinling? There's no better teacher than direct experience. You have already seen your first aftermarket GBA program running (that is, an unlicensed GBA game), the ShowPicture program from the previous chapter. I'm not going to open that project again, but you are free to return to the ShowPicture source code at any time as you increase your GBA coding skills (and you will explore bitmapped graphics extensively in Part Two). In the meantime, let's write a simple program that displays a message on the screen.

The important thing that I want you to grasp in this chapter is not the programming language used or the usefulness of a particular program, but rather just getting a good feel for how the development environment works; how to write code and compile your programs; and how to test programs using the emulator, flash linker, or multiboot cable. Therefore, I will start with a very simple program that just displays a message on the GBA screen. This first sample program that you will write uses the HAM library (Hamlib) to display text on the screen. Normally, something as seemingly simple as displaying a message would require a lot of work up front, because the GBA has no built-in function for drawing text on the screen. That's right, something as simple as a message requires a lot of work, which often involves creating a bitmap filled with font characters (or byte array of hard-coded letters) and writing a function to display a message using the font. It's all such an enormous amount of work for something so simple!

That is the way of things in console development—you have to do everything yourself. Even something as simple as polling a timer in order to maintain a constant refresh rate in a game is a very low-level activity, requiring intimate knowledge of how timers and interrupts work (see chapter 8, "Using Interrupts And Timers"). Now let's get started on the Greeting program.

Creating A New Project

If you haven't already, go ahead and fire up Visual HAM by double-clicking on the icon I helped you to create on your desktop or by browsing to the HAM installation folder and searching for VHAM.EXE. If you installed Visual HAM to the folder that I recommended, that might be located in C:\HAM\VHAM. Next, open the File menu and select New, then New Project, as shown in Figure 4.1.

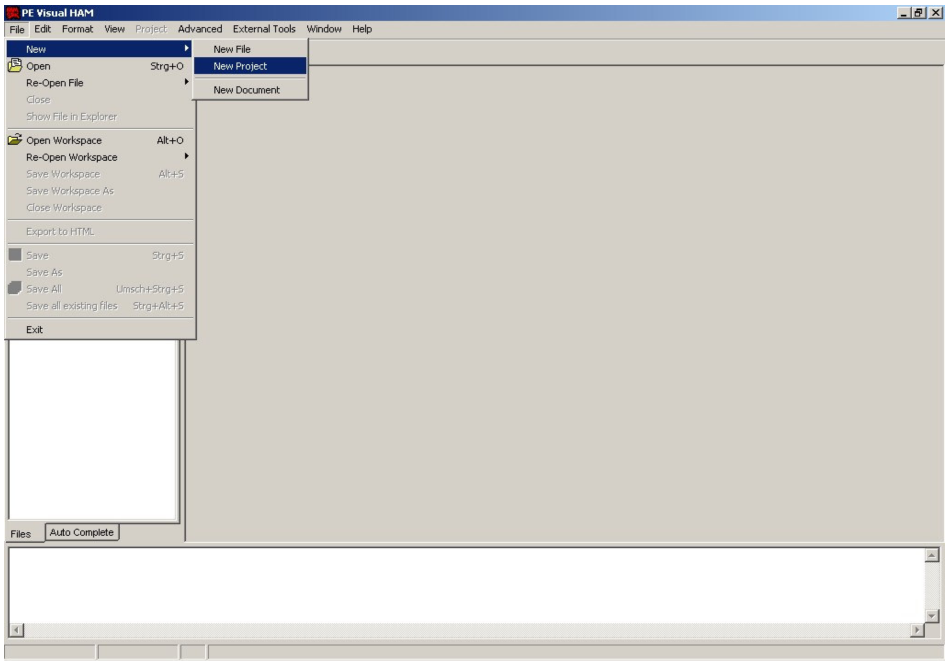
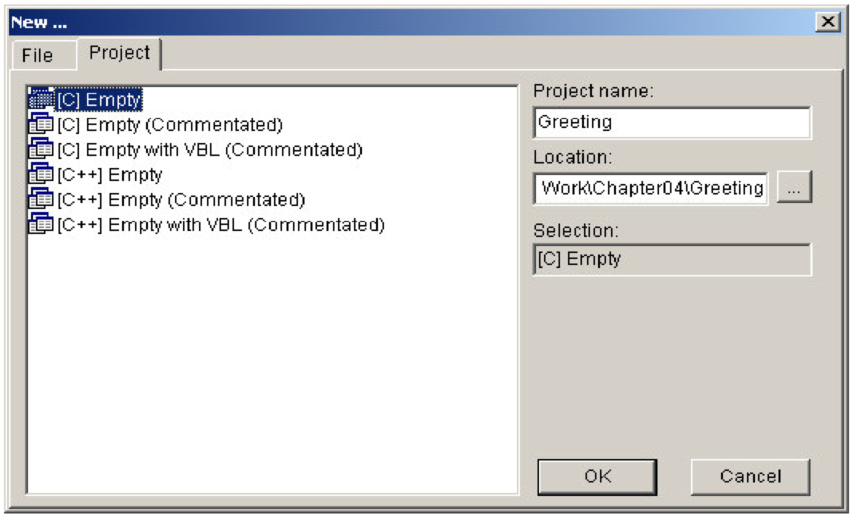


Figure 4.1

Creating a new project from the File menu of Visual HAM.

The New Project dialog box should appear, as shown in Figure 4.2. Select the first project type, [C] Empty, and then type "Greeting" for the project name. Select an appropriate location for this project on your hard disk drive. Click on the OK button to create the new project, and open the editor on the default source file.



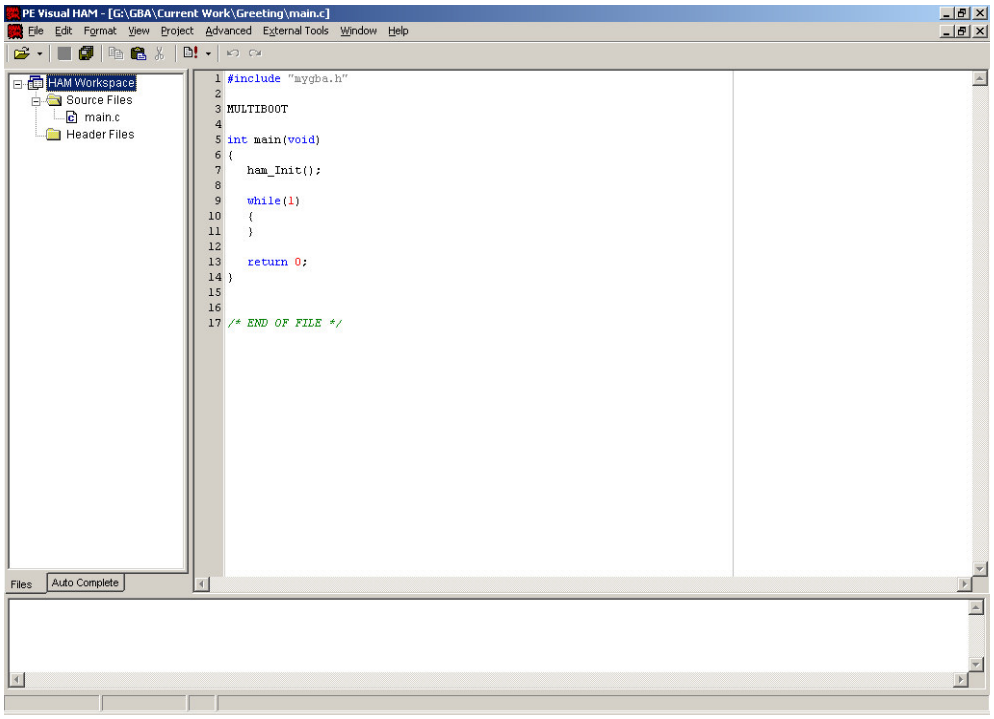
[C] Empty
[C] Empty (Commentated)
[C] Empty with VBL (Commentated)
[C++] Empty
[C++] Empty (Commentated)
[C++] Empty with VBL (Commentated)

Project name: Greeting
Location: Work\Chapter04\Greeting
Selection: [C] Empty

Figure 4.2

The New Project dialog box is where you specify the name, location, and type of project.

The source file for a new project in Visual HAM looks like the file shown in Figure 4.3. The default skeleton source code was already added to the main.c file for you by Visual HAM.


PE Visual HAM - [G:\GBA\Current Work\Greeting\main.c]
File Edit Format View Project Advanced External Tools Window Help
HAM Workspace
Source Files
main.c
Header Files
Files
Auto Complete

```
#include "mygba.h"

MULTIBOOT

int main(void)
{
   ham_Init();

   while(1)
   {
   }

   return 0;
}


/* END OF FILE */
```

Figure 4.3

A new project in Visual HAM with generated skeleton source code.

The source code looks like this:

```
#include "mygba.h"

MULTIBOOT

int main(void)
{
   ham_Init();
   while(1)
   {
   }

   return 0;
}

/* END OF FILE */
```

This default code is the minimum amount of code needed for a HAM program, and it actually *will* run (although it doesn't do anything useful). While the Greeting program you are about to write is indeed a HAM program (meaning that it uses the HAM library), there are many ways to write a GBA program, such as with straight C or C++, without any library at all. I will show you how to write a raw GBA program in the next section of this chapter (the next sample project, in fact). One interesting line is the MULTIBOOT statement. That is a specific statement that the GBA compiler recognizes and is somewhat like a macro of a define. It is needed when using the multiboot cable (more on that later in this chapter).

Writing The Greeting Program Source Code

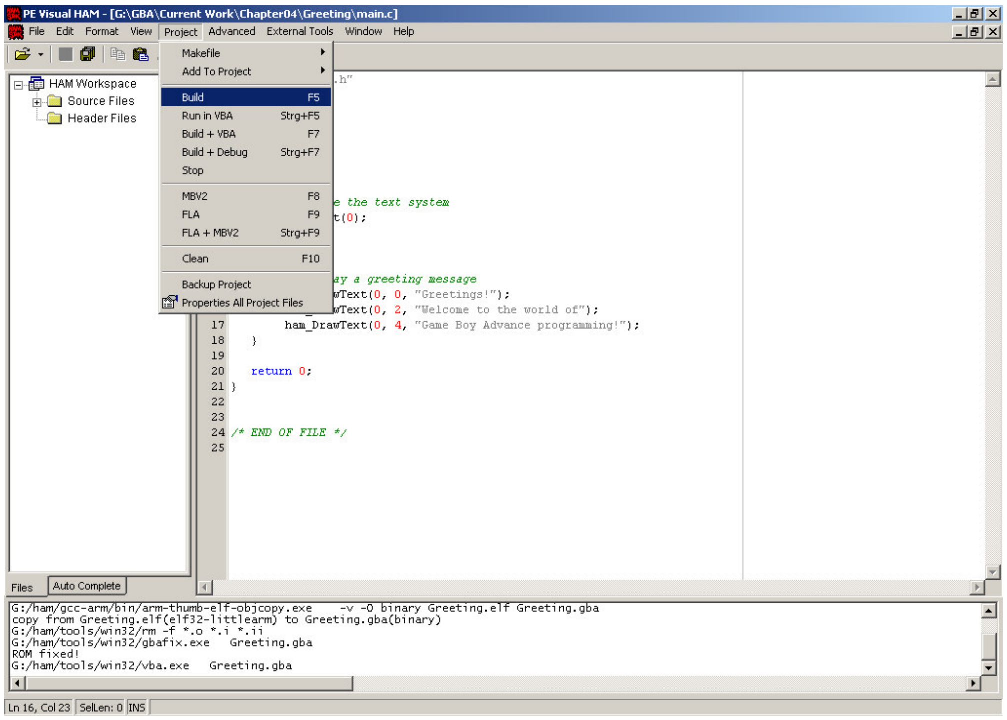
Okay, let's modify the skeleton program that Visual HAM generated for the Greeting project. There are two sections of code that I would like you to add to the program, as indicated in the code listing that follows. The new code is denoted in bold font. First, a line of code that initializes the Hamlib text system, and then three lines of code inside the while loop to actually display a message on the screen. Go ahead and modify the listing now as shown.

```
#include "mygba.h"

MULTIBOOT

int main(void)
{
    ham_Init();

    //initialize the text system
    ham_InitText(0);

    while(1)
    {
        //display a greeting message
        ham_DrawText(0, 0, "Greetings!");
        ham_DrawText(0, 2, "Welcome to the world of");
        ham_DrawText(0, 4, "Game Boy Advance programming!");
   }

   return 0;
}

/* END OF FILE */
```

Not bad, is it? The program is very short, and you should be able to easily understand what the program will do. It uses the ham_DrawText function to display a text string on the screen. Note that this function only works if you have first called ham_InitText, because that function loads the bitmap font used by the ham_DrawText function. It displays a nice system-type monospaced font that looks as if it were part of a built-in GBA text library, although you now know that Hamlib actually does all the work there!

Compiling The Greeting Program

Now let's compile the program. I will also show you some shortcut keys that you can use in Visual HAM to compile and run programs. If you look at Figure 4.4, it shows the Project menu in Visual HAM. Open the Project menu now, and select the Build menu item. This will invoke the compile process.



```
.h"

e the text system
t(0);

ay a greeting message
wText(0, 0, "Greetings!");
wText(0, 2, "Welcome to the world of");
        ham_DrawText(0, 4, "Game Boy Advance programming!");
    }

    return 0;
}


/* END OF FILE */
```

```
G:/ham/gcc-arm/bin/arm-thumb-elf-objcopy.exe    -v -O binary Greeting.elf Greeting.gba
copy from Greeting.elf(elf32-littlearm) to Greeting.gba(binary)
G:/ham/tools/win32/rm -f *.o *.i *.ii
G:/ham/tools/win32/gbafix.exe   Greeting.gba
ROM fixed!
G:/ham/tools/win32/vba.exe   Greeting.gba
```

Figure 4.4

The Project menu in Visual HAM, showing the Build menu item.

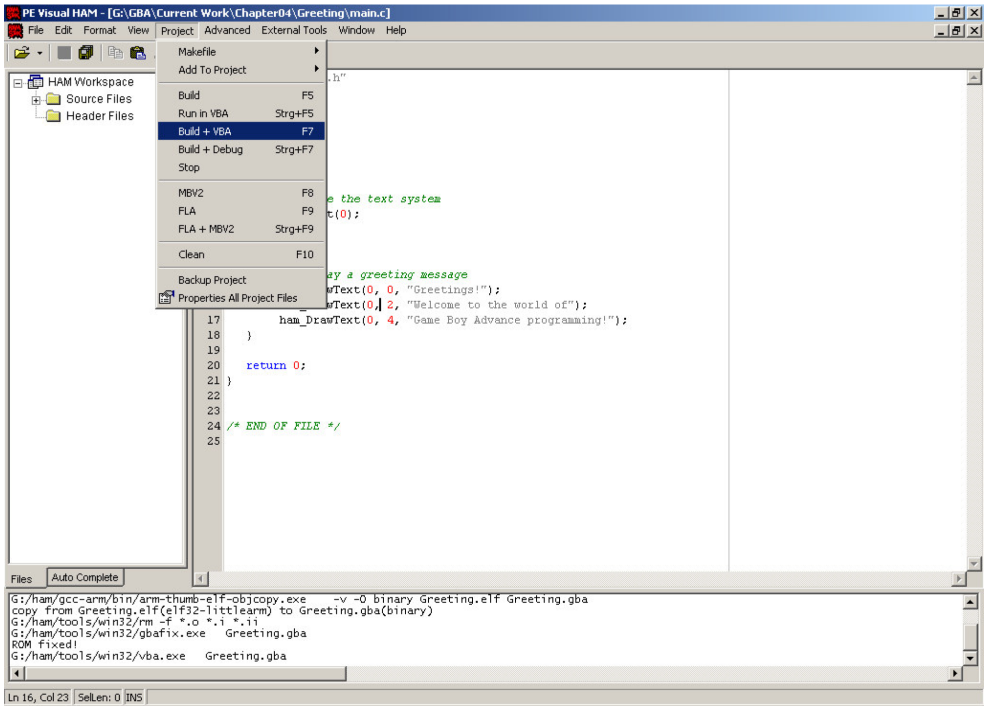
If the program has no syntax errors or typos, you should see no error messages appear in the output window down at the bottom of the screen in Visual HAM. When there is an error, it will be highlighted with a red "ERROR" message, which also displays the line number where the error occurred. I will get into debugging and error handling in later chapters. For now, if you see any error messages, the problem is most likely a typo, which you should be able to resolve by comparing the listing shown here with the source code on your screen. If there seems to be an error resulting from something other than a typo, it is possible that your installation of HAM is damaged, and you may want to refer back to the previous chapter to perform a reinstall of HAM. A common source of errors is when there are two different versions of HAM installed on your PC at the same time. Be sure to delete any older version before installing the latest version of HAM (which may be updated from the version included on the CD-ROM).

Press F5 to compile a project in Visual HAM.

Testing The Greeting Program In The Emulator

You are now ready to run the Greeting program on your PC using the emulator included with HAM, a program called VisualBoyAdvance. If you open the Project menu once again, look for the menu item called Build + VBA (as shown in Figure 4.5), and select it. The program should compile and begin running in the emulator, as shown in Figure 4.6.

Press F7 to compile the project and run it in the emulator.

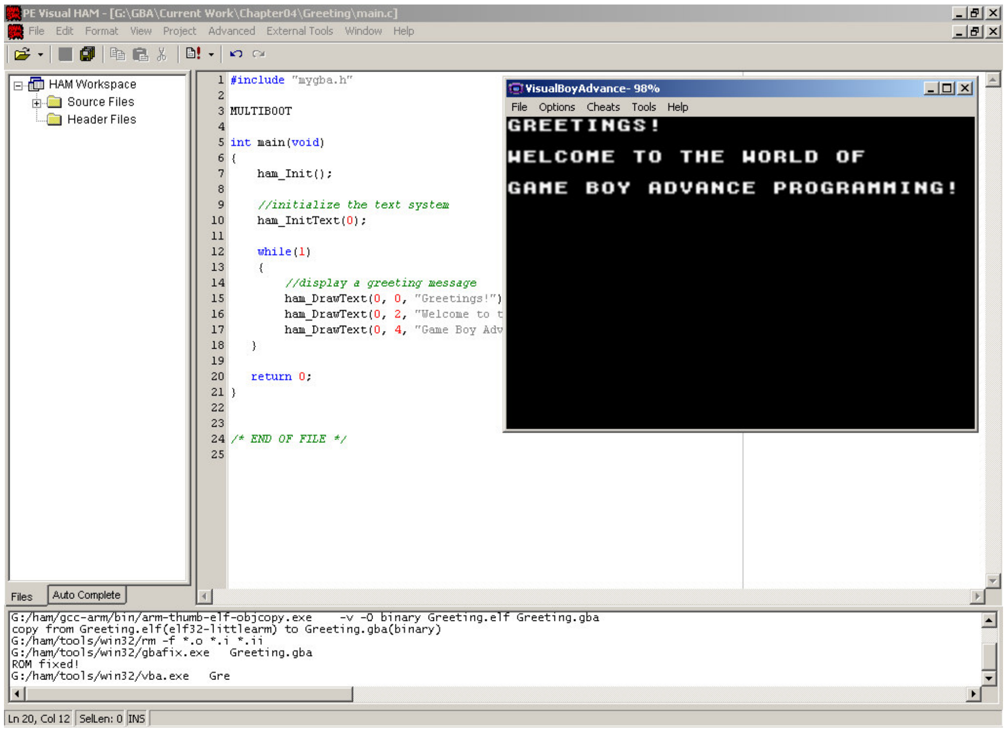


```
        ham_DrawText(0, 4, "Game Boy Advance programming!");
    }

    return 0;
}


/* END OF FILE */
```

```
G:/ham/gcc-arm/bin/arm-thumb-elf-objcopy.exe    -v -O binary Greeting.elf Greeting.gba
copy from Greeting.elf(elf32-littlearm) to Greeting.gba(binary)
G:/ham/tools/win32/rm -f *.o *.i *.ii
G:/ham/tools/win32/gbafix.exe   Greeting.gba
ROM fixed!
G:/ham/tools/win32/vba.exe   Greeting.gba
```

Figure 4.5

The Project menu, showing the Build + VBA menu item.


PE Visual HAM - [G:\GBA\Current Work\Chapter04\Greeting\main.c]
File Edit Format View Project Advanced External Tools Window Help
HAM Workspace
Source Files
Header Files
VisualBoyAdvance- 98%
File Options Cheats Tools Help
Files
Auto Complete
Ln 20, Col 12
SelLen: 0
INS

```
#include "mygba.h"

MULTIBOOT

int main(void)
{
    ham_Init();

    //initialize the text system
    ham_InitText(0);

    while(1)
    {
        //display a greeting message
        ham_DrawText(0, 0, "Greetings!")
        ham_DrawText(0, 2, "Welcome to t
        ham_DrawText(0, 4, "Game Boy Adv
    }

    return 0;
}


/* END OF FILE */
```

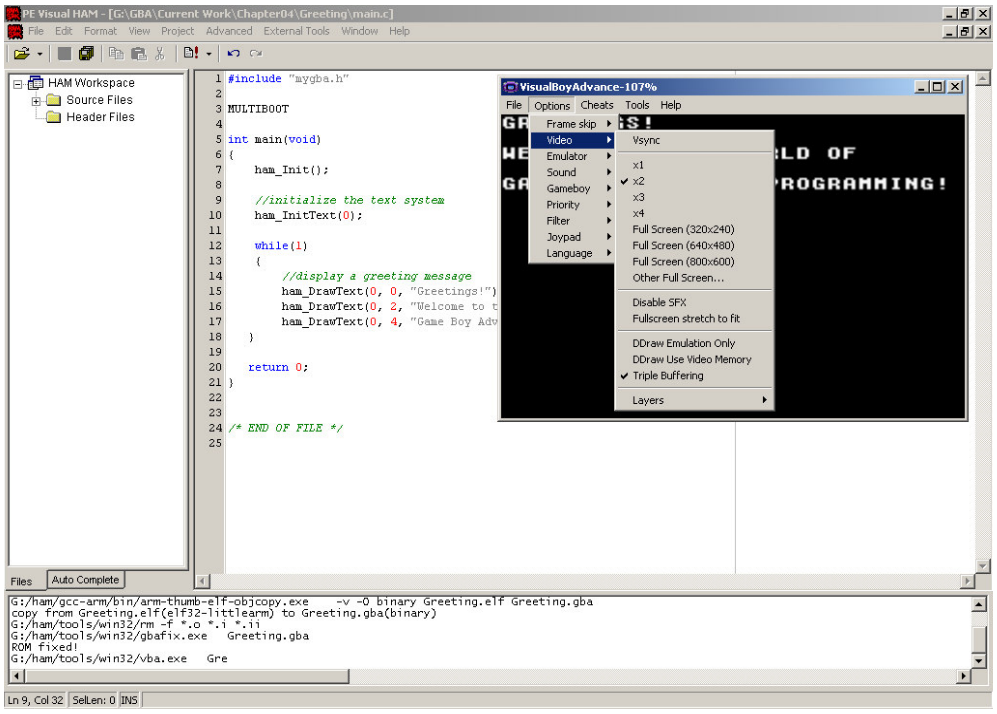
```
GREETINGS!

WELCOME TO THE WORLD OF

GAME BOY ADVANCE PROGRAMMING!
```

```
G:/ham/gcc-arm/bin/arm-thumb-elf-objcopy.exe    -v -O binary Greeting.elf Greeting.gba
copy from Greeting.elf(elf32-littlearm) to Greeting.gba(binary)
G:/ham/tools/win32/rm -f *.o *.i *.ii
G:/ham/tools/win32/gbafix.exe   Greeting.gba
ROM fixed!
G:/ham/tools/win32/vba.exe   Gre
```

Figure 4.6

The Greeting program running in the emulator.

In this screen shot, you can see that I have increased the default window size of VisualBoyAdvance to two times the normal size. You can change the size of the emulator window yourself by opening the Options menu (as shown in Figure 4.7), selecting Video, and then choosing a screen size for the emulator, from 1x up to 4x, and even one of several full-screen modes (which I don't recommend during development but encourage you to at least try out).



```
#include "mygba.h"

MULTIBOOT

int main(void)
{
    ham_Init();

    //initialize the text system
    ham_InitText(0);

    while(1)
    {
        //display a greeting message
        ham_DrawText(0, 0, "Greetings!")
        ham_DrawText(0, 2, "Welcome to t
        ham_DrawText(0, 4, "Game Boy Adv
    }

    return 0;
}


/* END OF FILE */
```

```
G:/ham/gcc-arm/bin/arm-thumb-elf-objcopy.exe    -v -O binary Greeting.elf Greeting.gba
copy from Greeting.elf(elf32-littlearm) to Greeting.gba(binary)
G:/ham/tools/win32/rm -f *.o *.i *.ii
G:/ham/tools/win32/gbafix.exe   Greeting.gba
ROM fixed!
G:/ham/tools/win32/vba.exe   Gre
```

Figure 4.7

Changing the window size of the emulator using the Options menu.

Drawing Pixels

Now for something really fun! This program might surprise you. After all, it's a real GBA program, and *yes*, it will run on an actual GBA (using a flash linker, for instance). The great thing about this program is just how small it is. Now, I realize this doesn't do much, but it's a 100 percent bona fide GBA program, and it does one of the most basic things that you must learn when programming video games—drawing pixels. This is the basis for all video games! Drawing a single pixel in video game terms is sort of like the Hello World mantra established by Brian W. Kernighan and Dennis M. Ritchie in their famous book *The C Programming Language* (the book that first introduced the world to the C language—the language used in this book). The Greeting program you just wrote was a sort of Hello World program, but it used Hamlib (because, as I explained, it would be too difficult to display text without Hamlib at this point).

Now, assuming Visual HAM is still running from the last project, what you will want to do is create a new project. Alternatively, you may load the project from the CD-ROM (\Sources\Chapter04\); you can do this for any of the projects in this chapter. But where's the fun in that? This program is little, so I insist you type it in! However, for future reference, note that Visual HAM project files have an extension of .vhw. So, anytime you need to open a GBA project from within Windows Explorer, you can simply double-click the .vhw file (in this case, DrawPixel.vhw). There is usually also a binary executable file with an extension of .gba along with each project. Double-clicking the .gba file should cause the emulator to start. If it doesn't, simply locate the VisualBoyAdvance.exe (or vba.exe) program file on your hard drive in order to associate .gba files with the emulator.

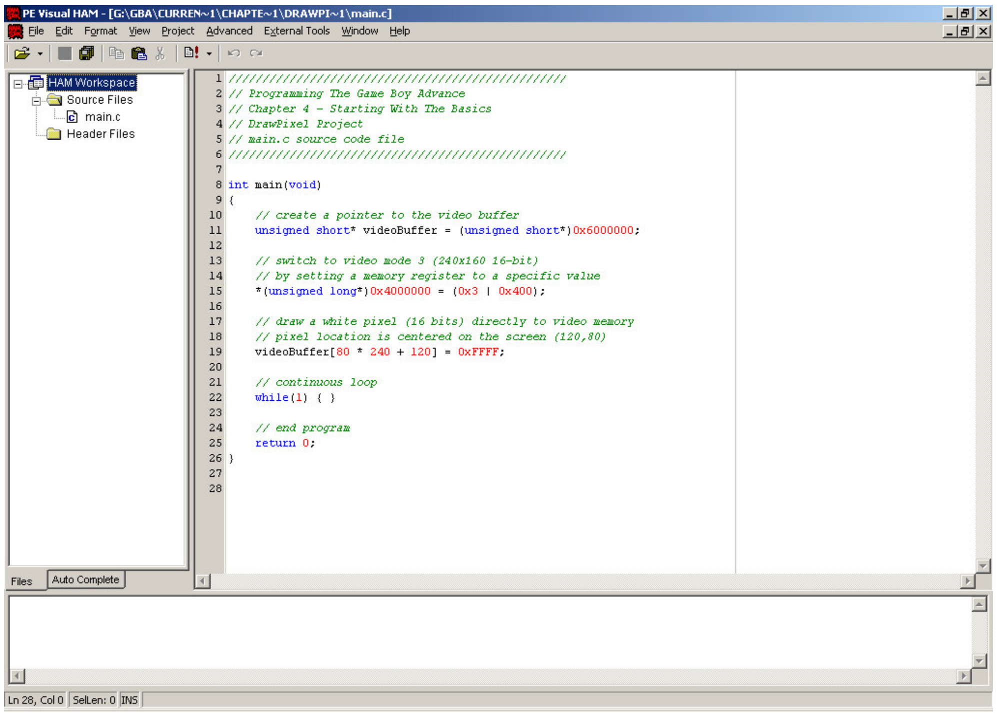
Create a new blank project and delete whatever default code Visual HAM fills in automatically. We're going to start from scratch here. Name the project DrawPixel and give it a new project folder (which is created by Visual HAM). If you are having trouble creating the project, refer to the figures from the previous sample program to refresh your memory.

Writing the DrawPixel Program Source Code

Let me first explain this code a little. I left it intentionally sparse in order to make a point, that a GBA program can be very small, and small it is indeed! I'm not sure if it's possible to write a smaller GBA program than this (excluding the comments, of course). It is really only . . . let's see . . . eight lines of code, counting the curly brackets. Okay, go ahead and type it in. I'll explain what is going on in this program after the listing.

```
//////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 4: Starting With The Basics
// DrawPixel Project
// main.c source code file
//////////////////////////////////////////////////

int main(void)
{
    // create a pointer to the video buffer
    unsigned short* videoBuffer = (unsigned short*)0x6000000;
    // switch to video mode 3 (240x160 16-bit)
    // by setting a memory register to a specific value
    *(unsigned long*)0x4000000 = (0x3 | 0x400);

    // draw a white pixel (16 bits) directly to video memory
    // pixel location is centered on the screen (120,80)
    videoBuffer[80 * 240 + 120] = 0xFFFF;

    // continuous loop
    while(1) { }

    // end program
    return 0;
}
```

After you have finished typing in the code, the IDE should look like Figure 4.8.


PE Visual HAM - [G:\GBA\CURREN~1\CHAPTE~1\DRAWPI~1\main.c]
File Edit Format View Project Advanced External Tools Window Help
HAM Workspace
Source Files
main.c
Header Files
Files
Auto Complete
Ln 28, Col 0
SelLen: 0
INS

```
//////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 4 - Starting With The Basics
// DrawPixel Project
// main.c source code file
//////////////////////////////////////////////

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

    // switch to video mode 3 (240x160 16-bit)
    // by setting a memory register to a specific value
    *(unsigned long*)0x4000000 = (0x3 | 0x400);

    // draw a white pixel (16 bits) directly to video memory
    // pixel location is centered on the screen (120,80)
    videoBuffer[80 * 240 + 120] = 0xFFFF;

    // continuous loop
    while(1) { }

    // end program
    return 0;
}
```

Figure 4.8
The DrawPixel project as it appears in the Visual HAM IDE.

This program jumps ahead a bit, because graphics are really covered in Part II, starting with the next chapter. But I wanted to give you a taste of what is to come. This is a short program, but it provides a basis for just about any GBA game you are likely to write in the future.

Are you surprised to find no includes in this program? If you are an old hand with C, you are likely wondering why there are no header files for interfacing with the GBA. That's the beauty of the HAM distribution and the Visual HAM IDE. There are really no headers included by default, and none are needed for the most basic programs, although the HAM SDK includes several libraries automatically when the program is compiled and linked into an exe file. You will face this situation throughout the book. The GBA uses memory registers to perform basic functions. For instance, the code that sets the video mode:

```
*(unsigned long*)0x4000000 = (0x3 | 0x400);
```

is just a pointer to a memory location, and a specific numeric value is set in that memory location. In this instance, 0x3 is video mode 3: 240 x 160, 16-bit, while 0x400 refers to background 2, and these values are combined with a bitwise OR (the pipe symbol, |). I will explain these things in more detail in Part Two, which is dedicated entirely to graphics programming. For now, the point is to get a feel for *what* a GBA program is like, rather than specifically *how* every line of code works.

The next non-comment line of code:

```
videoBuffer[80 * 240 + 120] = 0xFFFF;
```

actually draws the pixel at the center of the screen. Video mode 3 has a resolution of 240 x 160, so the center of the screen is at 120 x 80. The formula for accessing a linear memory array using two-dimensional coordinates (in this case, the pixel's X,Y location) is this:

```
Memory Location = Y * Screen Width + X
```

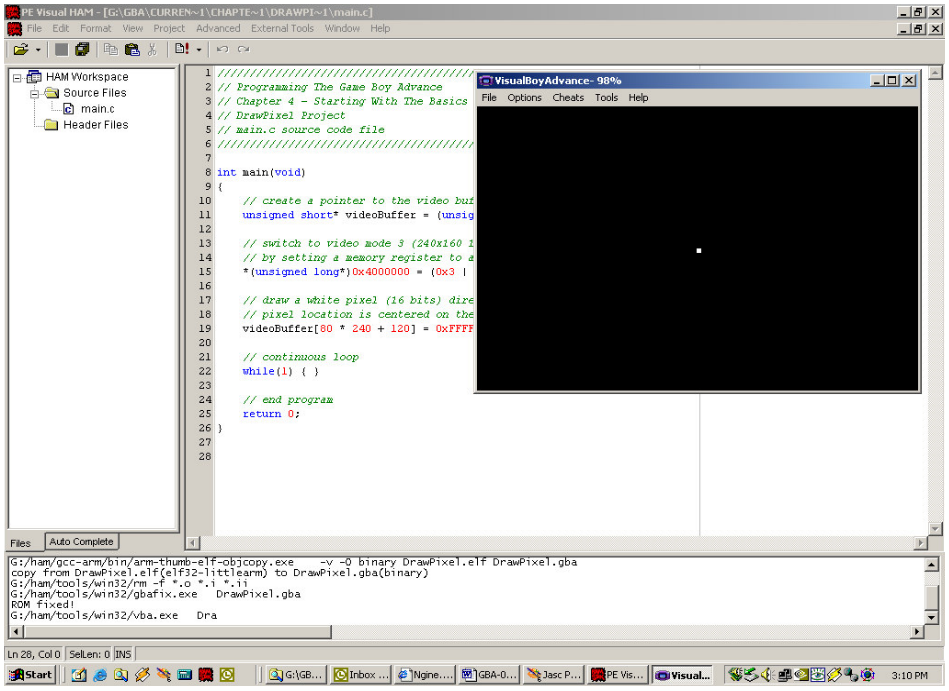
By filling in this memory location formula for the location in the video buffer, you are able to then set that memory location to a specific value—the color of the pixel, which was set to 0xFFFF in this case. 0xFFFF is a hexadecimal number, where each character after the 0x is 4 bits in size (with possible values of 0–9 and A–F, for a total of 16 bits). Therefore, 0xFFFF is a 16-bit number, which is exactly what is needed for mode 3, because it uses 16-bit pixels. In the graphics chapters of Part II, I will explain how to set the pixel color using the usual (Red,Green,Blue) components—which is somewhat beyond the scope of this short example.

Compiling the DrawPixel Program

Now, I'm sure you have already jumped ahead and run the program, per the instructions provided in the previous sample program. If you have not already done so, go ahead and compile (or rather, build) the program by pressing F5. If all goes well and there are no syntax errors in your program, then it is ready to be run in the emulator.

Testing the DrawPixel Program in the Emulator

At this point, you may open the Project menu and select Build + VBA to run the program (or simply press F7 to perform this step with a single keystroke). The running DrawPixel program is shown in Figure 4.9. Do you see the small pixel in the center of the emulator window?


PE Visual HAM - [G:\GBA\CURREN~1\CHAPTE~1\DRAWPI~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- 98%
File Options Cheats Tools Help
Ln 28, Col 0
SelLen: 0
INS
Start

```
//////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 4 - Starting With The Basics
// DrawPixel Project
// main.c source code file
//////////////////////////////////////////

int main(void)
{
    // create a pointer to the video buf
    unsigned short* videoBuffer = (unsig

    // switch to video mode 3 (240x160 1
    // by setting a memory register to a
    *(unsigned long*)0x4000000 = (0x3 |

    // draw a white pixel (16 bits) dire
    // pixel location is centered on the
    videoBuffer[80 * 240 + 120] = 0xFFFF

    // continuous loop
    while(1) { }

    // end program
    return 0;
}
```

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

Figure 4.9

The DrawPixel program draws a single pixel in the center of the GBA screen.

Filling The Screen

Now, after seeing just one pixel on the screen, I can't resist the temptation to take it a step further and fill the entire screen with pixels! This code is just starting to become fun to write, so let's go for it and write another graphics program. I know, this is all covered in Part Two, as I have explained already, but it couldn't hurt to take a peek a bit early, right?

Writing the FillScreen Program Source Code

You should be pretty good at creating a new project in Visual HAM after the last two sample programs, so I won't go over that again right now (although I will go through the process at least once in each chapter, lest you get lost at any point). This program, which is called FillScreen, uses some defines and one function, simply because it's too hard to remember the special memory addresses of the GBA from memory (how's that for a tongue twister?). Advanced GBA projects use an include file with all of the memory addresses and registers in the GBA (see Appendix C, "Game Boy Advance Hardware Reference"), so I might as well give you a sneak peek at what some of those things look like. This program creates a define for the memory register for changing video modes, which you saw in the previous sample program. The register is actually called REG_DISPCNT, and the define looks like this:

```
#define REG_DISPCNT *(unsigned long*)0x4000000
```

In case you aren't a C guru (and I'm not expecting you to be, although I know some folks out there are old hands with C), the #define statement allows you to create a macro that the compiler fills in at compile time. What this means is that anytime the compiler finds REG_DISPCNT in the source code, it fills in *(unsigned long*)0x4000000 in its place! There's no denying that this little feature will preserve your sanity, because there are many, many memory registers in the GBA architecture that resemble this particular one. What happens, specifically, is that when the GBA detects a change at that memory address, it knows that it should change video modes to the number specified. This is very much like a function call, as if there is a sort of SetMode function built into the GBA. While it isn't called SetMode, the memory register is essentially the same thing. When you set the memory register to a specific value, such as 0x3, you are actually passing a parameter to the "function", so to speak. Are you following me? If not, that's okay, because I'll explain each new memory register as needed in later chapters, so you'll get the hang of it in time. I will admit, this is a new and foreign way to write code, especially for those of us who are used to procedural or object-oriented programming. But that is what makes console coding so rewarding—you are closer to the hardware and actually manipulating physical parts of the memory built into the GBA, in order to do things.

There are three more defines in this program as well. The next one:

```
#define MODE_3 0x3
```

you may recognize from the DrawPixel program. This is the video mode that the program uses, mode 3, with a resolution of 240 x 160 and 16-bit color depth. By defining it to MODE_3, it's easier to remember exactly what mode the program is using. I realize that 0x3 is easy to spot as well, so if you prefer that, go ahead and use the literal instead of the define (that is what I do in later chapters).

The next define:

```
#define BG2_ENABLE 0x400
```

is also related to the video mode. As you may recall, the DrawPixel program set the video mode by OR'ing 0x3 with 0x400, in order to specify that the program should use background 2. This is something that I will cover, again, in Part Two, so don't worry about it at this point.

The last define:

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

creates a macro for packing an RGB color into a 16-bit value. This allows you to pass parameters to the define, as if it were a function. In fact, this could be written as a function instead of as a define, but the define is simpler.

Finally, the DrawPixel3 function:

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

This function, as the name implies, draws a pixel on the mode 3 screen. The single line of code in this function also resembles the code in the DrawPixel program, but this version now allows you to pass the X,Y values as parameters. It is, therefore, more useful as a function. Now here is the complete source code listing:

```
//////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 4 - Starting With The Basics
// FillScreen Project
// main.c source code file
//////////////////////////////////////////////////

//define register for changing the video mode
#define REG_DISPCNT *(unsigned long*)0x4000000

//video mode 3 = 240x160 16-bit
#define MODE_3 0x3

//use background 2
#define BG2_ENABLE 0x400

//macro to pack an RGB color into 16 bits
#define RGB(r,g,b) (unsigned short)(r + (g << 5) + (b << 10))

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

//draw a pixel on the mode 3 video buffer
void DrawPixel3(int x, int y, unsigned short c)
{
    videoBuffer[y * 240 + x] = c;
}

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

    //switch to video mode 3 (240x160 16-bit)
    REG_DISPCNT = (MODE_3 | BG2_ENABLE);
    //fill the screen
    for (x = 0; x < 239; x++)
    {
        for (y = 0; y < 159; y++)
        {
            DrawPixel3(x, y, RGB(0,(255-y),x));
        }
    }

    // continuous loop
    while(1)
    {
        // do nothing
    }

    // end program
    return 0;
}
```

The key to this program is the section of code denoted by the comment "fill the screen". Here are two for loops: the first for the X values, the second for the Y values. Inside the loops is a call to DrawPixel3 with the X and Y variables. A creative use of the Y value provides the color, thus filling in the screen with an interesting fill effect.

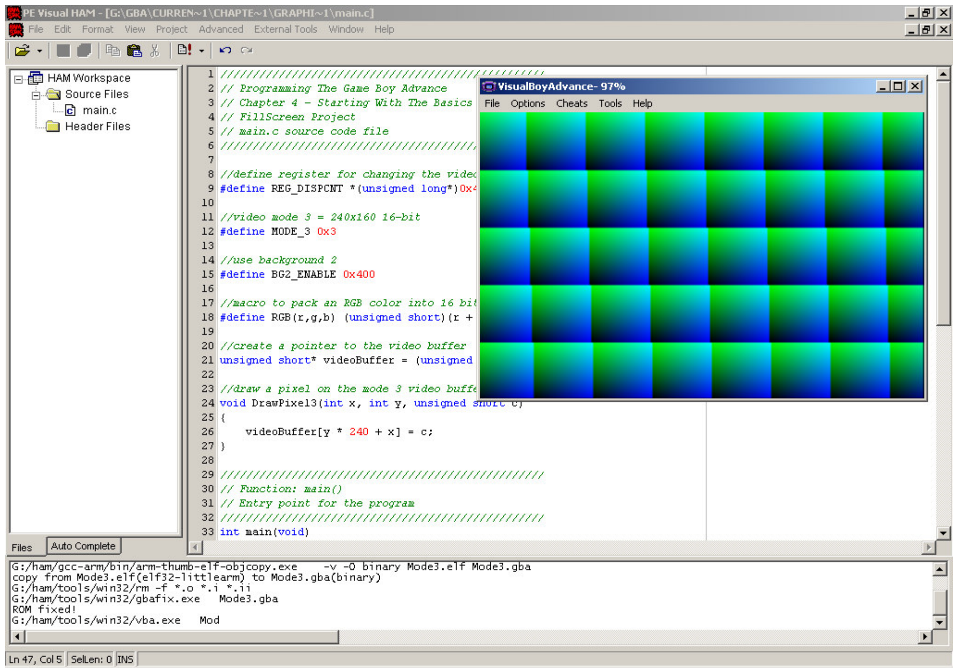
Compiling the FillScreen Program

Now, go ahead and compile the program by pressing F5. This is the most complicated program you have written so far, so don't be surprised if there are a few syntax errors. The most common errors involve a missing semicolon at the end of a line or a missing closing curly brace at the end of a block of code (such as with a for loop). If there are any errors, closely examine the source code on your screen and compare it with the printed listing to locate errors. Another potential source of errors is the case of variable names. Remember that in C, everything is case sensitive, so that X is recognized as a different variable than x. This can be very confusing at times, so take care to watch the case when naming variables.

Testing the FillScreen Program in the Emulator

Now, one of the reasons why I include the compile step separately from the testing, or running, step here is to make sure the program works first. Obviously, after you are more experienced with Visual HAM and have been working on a program for hours, you will likely just skip the compile step and go directly to the Build + VBA step by pressing F7. This is what I usually do after the first few times. I often first compile a program when loading up someone *else's* game (because there are a lot of public domain GBA games available online—see Appendix B for a list of good Web sites featuring fan-written GBA games). HAM is capable of running programs not even created under Visual HAM, because it uses the same GCC compiler that the other GBA development kits use. However, that may not always be the case, as new development kits are appearing all the time; such is the case in the open source community. Usually, though, most programmers use the top one or two development kits, and HAM is definitely one of those.

Go ahead and run the program now. If all goes well, you should see the emulator window appear with a colorful pattern filling the simulated GBA screen, as shown in Figure 4.10.


PE Visual HAM - [G:\GBA\CURREN~1\CHAPTE~1\GRAPHI~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- 97%
File Options Cheats Tools Help
Ln 47, Col 5 SelLen: 0 INS

```
///////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 4 - Starting With The Basics
// FillScreen Project
// main.c source code file
////////////////////////////////////////////

//define register for changing the videc
#define REG_DISPCNT *(unsigned long*)0x4

//video mode 3 = 240x160 16-bit
#define MODE_3 0x3

//use background 2
#define BG2_ENABLE 0x400

//macro to pack an RGB color into 16 bit
#define RGB(r,g,b) (unsigned short)(r +

//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned

//draw a pixel on the mode 3 video buffe
void DrawPixel3(int x, int y, unsigned short c)
{
    videoBuffer[y * 240 + x] = c;
}

///////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
///////////////////////////////////////////////////////////
int main(void)
```

```
G:/ham/gcc-arm/bin/arm-thumb-elf-objcopy.exe    -v -O binary Mode3.elf Mode3.gba
copy from Mode3.elf(elf32-littlearm) to Mode3.gba(binary)
G:/ham/tools/win32/rm -f *.o *.i *.ii
G:/ham/tools/win32/gbafix.exe   Mode3.gba
ROM fixed!
G:/ham/tools/win32/vba.exe   Mod
```

Figure 4.10

The FillScreen program fills the GBA screen with a pattern of pixels.

You know, with the source code for FillScreen, you have everything you need to write a rudimentary GBA game at this point. That is amazing considering that you were only drawing your first pixel a short while ago. But the fact remains that once you are able to draw a pixel, it follows that you are able to write a game with only a little more effort.

Such a game might not have advanced, high-speed blitting (a fast method of drawing graphic images) or transparency, but it is still a significant possibility.

But wait, something is missing. First, you need to be able to capture button presses! At present, all you can really do is write a demo—something interesting graphically, but with no possibility for input. <Sigh>. Okay, there's still a lot of ground to cover before writing your first game, but I don't want to discourage you. Therefore, let's take a quick tutorial on reading button presses on the GBA.

If you are at all excited about these basics, wait until you get to Chapter 7, "Rounding Up Sprites," where you'll learn how to use hardware-accelerated sprites with built-in transparency, alpha-blending, rotation, and scaling capabilities! Not only that, in Chapter 6, "Tile-Based Video Modes" will teach you how to create scrolling backgrounds. By the time you have finished those chapters, you will have no need for pixels at all. But it's nice to start with the basics, because that helps ones to appreciate what the GBA can do.

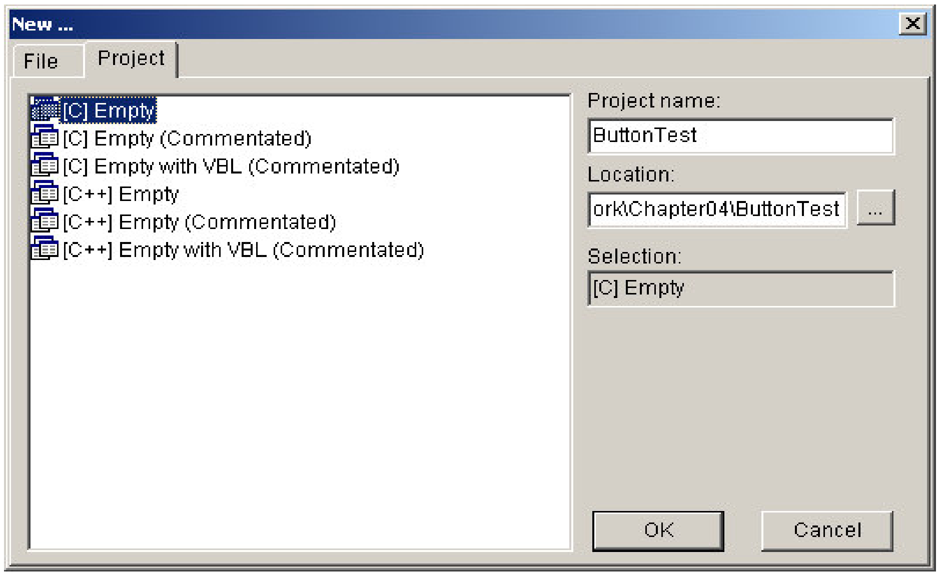
Detecting Button Presses

This section includes a program called ButtonTest that—surprise!—detects the GBA buttons. Because this is such a significant part of gameplay, and the subject isn't covered fully until Chapter 10, "Interfacing With the Buttons," I wanted to give you a little exposure to button programming at this point. I know you will have some fun with the code presented in this program! The ButtonTest program uses the ham_DrawText function to display text messages on the screen in order to update the status of each button, which appears by name on the screen (with a small "x"). The button presses are detected by a series of if . . . else statements and button macros such as F_CTRLINPUT_UP_PRESSED (which is specific to Hamlib).

Writing the ButtonTest Program Source Code

The source code for the ButtonTest program is about a page and a half in length, and some of it comprises comments (which you may leave out, if you wish). Basically, this program is a simple loop with a continuous conditional check of the buttons on the GBA using a series of if . . . else statements. First, the program initializes HAM by calling ham_Init, which must be called before using any of the features of the HAM library. Next, ham_InitText(0) is called to initialize the text mode of Hamlib using a specified background number (I will talk more about backgrounds in Part Two). After these two initializing functions have been called, the program uses ham_DrawText to display the status of each button on the screen.

First, you need to create a new project called ButtonTest, using the same procedure you have followed for the last three sample programs. Fire up Visual HAM, open the File menu, and select New, and then New Project to open the New Project dialog box, as shown in Figure 4.11. Select the [C] Empty project type. For the project name, type in "ButtonTest", and then type in the folder where you would like the project to be created. Note that Visual HAM will create the folder if it doesn't already exist.


New ...
File
Project
[C] Empty
[C] Empty (Commentated)
[C] Empty with VBL (Commentated)
[C++] Empty
[C++] Empty (Commentated)
[C++] Empty with VBL (Commentated)
...
OK
Cancel

Project name: ButtonTest
Location: ork\Chapter04\ButtonTest
Selection: [C] Empty

Figure 4.11
The New Project dialog box.

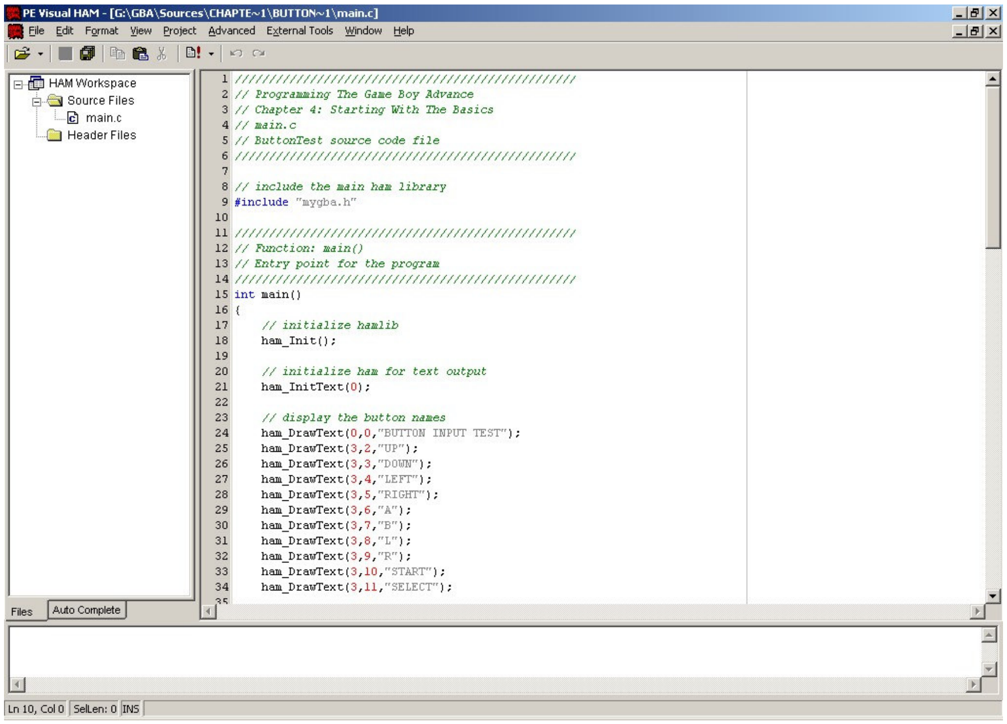
Next, just delete the skeleton code Visual HAM generated for you, and type in the following code listing for the ButtonTest program. Or, if you are good at filling in the details, you may simply type this program into the skeleton code, filling in where necessary, because the generated code is included in this listing. Just be sure not to leave out anything, as it's easy to lose your place while filling in code (just as it's easy to lose your place when typing in an entire code listing from scratch).

I emphasize typing because there truly is no better way to familiarize yourself with a new programming language or SDK. If you simply load up each of the sample programs from the CD-ROM, you may run them and see what the programs look like. However, you lose that critical step—typing in the code makes you intimately familiar with the function calls and gives you deeper insight into how the program works.

Here is the complete listing for the ButtonTest program:

```
//////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 4: Starting With The Basics
// ButtonTest Project
// main.c source code file
//////////////////////////////////////////////////

// include the main ham library
#include "mygba.h"

// enable multi-boot support
MULTIBOOT

//////////////////////////////////////////////////
// Function: main()
// Entry point for the program
//////////////////////////////////////////////////
int main()
{
    // initialize hamlib
    ham_Init();

    // initialize ham for text output
    ham_InitText(0);

    // display the button names
    ham_DrawText(0,0,"BUTTON INPUT TEST");
    ham_DrawText(3,2,"UP");
    ham_DrawText(3,3,"DOWN");
    ham_DrawText(3,4,"LEFT");
    ham_DrawText(3,5,"RIGHT");
    ham_DrawText(3,6,"A");
    ham_DrawText(3,7,"B");
    ham_DrawText(3,8,"L");
    ham_DrawText(3,9,"R");
    ham_DrawText(3,10,"START");
    ham_DrawText(3,11,"SELECT");

    // continuous loop
    while(1)
    {
        // check UP button
        if (F_CTRLINPUT_UP_PRESSED)
            ham_DrawText(0,2,"X");
        else
            ham_DrawText(0,2," ");

        // check DOWN button
        if (F_CTRLINPUT_DOWN_PRESSED)
            ham_DrawText(0,3,"X");
        else
            ham_DrawText(0,3," ");

        // check LEFT button
        if (F_CTRLINPUT_LEFT_PRESSED)
            ham_DrawText(0,4,"X");
        else
            ham_DrawText(0,4," ");

        // check RIGHT button
        if (F_CTRLINPUT_RIGHT_PRESSED)
            ham_DrawText(0,5,"X");
        else
            ham_DrawText(0,5," ");

        // check A button
        if (F_CTRLINPUT_A_PRESSED)
            ham_DrawText(0,6,"X");
        else
            ham_DrawText(0,6," ");

        // check B button
        if (F_CTRLINPUT_B_PRESSED)
            ham_DrawText(0,7,"X");
        else
            ham_DrawText(0,7," ");

        // check L button
        if (F_CTRLINPUT_L_PRESSED)
            ham_DrawText(0,8,"X");
        else
            ham_DrawText(0,8," ");

        // check R button
        if (F_CTRLINPUT_R_PRESSED)
            ham_DrawText(0,9,"X");
        else
            ham_DrawText(0,9," ");

        // check START button
        if (F_CTRLINPUT_START_PRESSED)
            ham_DrawText(0,10,"X");
        else
            ham_DrawText(0,10," ");

        // check SELECT button
        if (F_CTRLINPUT_SELECT_PRESSED)
            ham_DrawText(0,11,"X");
        else
            ham_DrawText(0,11," ");
    }
    // end program
    return 0;
}
```

After you have finished typing in the source code for the ButtonTest program, the editor should look something like Figure 4.12.


PE Visual HAM - [G:\GBA\Sources\CHAPTE~1\BUTTON~1\main.c]
File Edit Format View Project Advanced External Tools Window Help
HAM Workspace
Source Files
main.c
Header Files
Files
Auto Complete
Ln 10, Col 0
SelLen: 0
INS

```
//////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 4: Starting With The Basics
// main.c
// ButtonTest source code file
//////////////////////////////////////////////////

// include the main ham library
#include "mygba.h"

//////////////////////////////////////////////////
// Function: main()
// Entry point for the program
//////////////////////////////////////////////////
int main()
{
    // initialize hamlib
    ham_Init();

    // initialize ham for text output
    ham_InitText(0);

    // display the button names
    ham_DrawText(0,0,"BUTTON INPUT TEST");
    ham_DrawText(3,2,"UP");
    ham_DrawText(3,3,"DOWN");
    ham_DrawText(3,4,"LEFT");
    ham_DrawText(3,5,"RIGHT");
    ham_DrawText(3,6,"A");
    ham_DrawText(3,7,"B");
    ham_DrawText(3,8,"L");
    ham_DrawText(3,9,"R");
    ham_DrawText(3,10,"START");
    ham_DrawText(3,11,"SELECT");
```

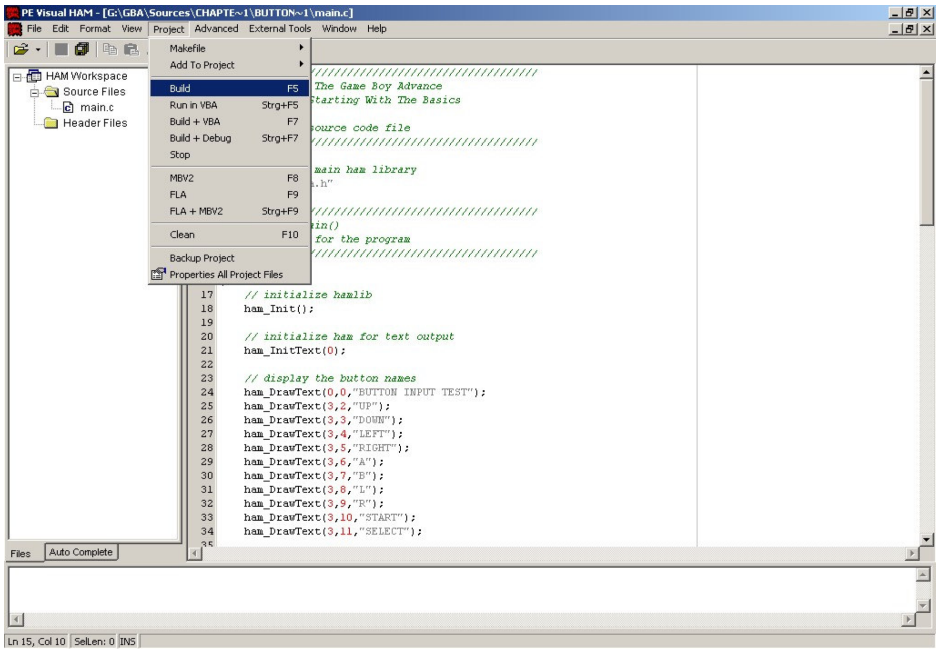
Figure 4.12

The ButtonTest program demonstrates the button handler built into Hamlib.

Compiling the ButtonTest Program

Now let's compile the ButtonTest program. Just for reference, because it's been a while, I'll go through the steps with you again. First, open the Project menu and select Build, as shown in Figure 4.13.

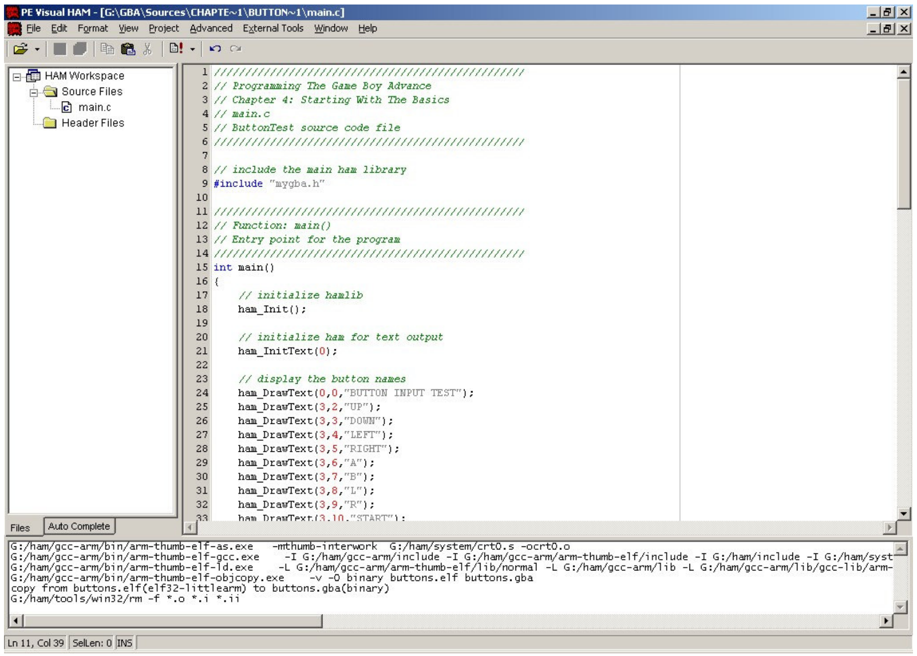
Now for a little more detail as to what is happening. The build command invokes the make utility to run the makefile that is generated by Visual HAM when you created the project. After invoking a build, the IDE looks like Figure 4.14. Note the compiler messages at the bottom of the screen.



```
    // initialize hamlib
    ham_Init();

    // initialize ham for text output
    ham_InitText(0);

    // display the button names
    ham_DrawText(0,0,"BUTTON INPUT TEST");
    ham_DrawText(3,2,"UP");
    ham_DrawText(3,3,"DOWN");
    ham_DrawText(3,4,"LEFT");
    ham_DrawText(3,5,"RIGHT");
    ham_DrawText(3,6,"A");
    ham_DrawText(3,7,"B");
    ham_DrawText(3,8,"L");
    ham_DrawText(3,9,"R");
    ham_DrawText(3,10,"START");
    ham_DrawText(3,11,"SELECT");
```

Figure 4.13

Use the Project, Build menu to compile a program in Visual HAM.


PE Visual HAM - [G:\GBA\Sources\CHAPTE~1\BUTTON~1\main.c]
File Edit Format View Project Advanced External Tools Window Help
HAM Workspace
Source Files
main.c
Header Files
Files
Auto Complete
Ln 11, Col 39 SelLen: 0 INS

```
/////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 4: Starting With The Basics
// main.c
// ButtonTest source code file
/////////////////////////////////////////////////

// include the main ham library
#include "mygba.h"

/////////////////////////////////////////////////
// Function: main()
// Entry point for the program
/////////////////////////////////////////////////
int main()
{
    // initialize hamlib
    ham_Init();

    // initialize ham for text output
    ham_InitText(0);

    // display the button names
    ham_DrawText(0,0,"BUTTON INPUT TEST");
    ham_DrawText(3,2,"UP");
    ham_DrawText(3,3,"DOWN");
    ham_DrawText(3,4,"LEFT");
    ham_DrawText(3,5,"RIGHT");
    ham_DrawText(3,6,"A");
    ham_DrawText(3,7,"B");
    ham_DrawText(3,8,"L");
    ham_DrawText(3,9,"R");
    ham_DrawText(3,10,"START");
```

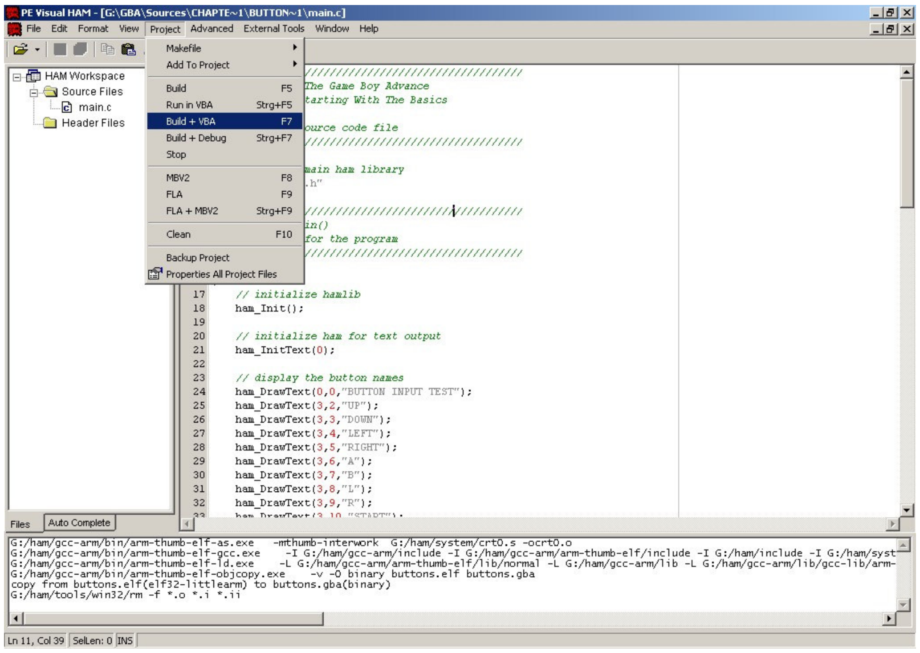
```
G:/ham/gcc-arm/bin/arm-thumb-elf-as.exe   -mthumb-interwork  G:/ham/system/crt0.s -ocrt0.o
G:/ham/gcc-arm/bin/arm-thumb-elf-gcc.exe    -I G:/ham/gcc-arm/include -I G:/ham/gcc-arm/arm-thumb-elf/include -I G:/ham/include -I G:/ham/syst
G:/ham/gcc-arm/bin/arm-thumb-elf-ld.exe    -L G:/ham/gcc-arm/arm-thumb-elf/lib/normal -L G:/ham/gcc-arm/lib -L G:/ham/gcc-arm/lib/gcc-lib/arm-
G:/ham/gcc-arm/bin/arm-thumb-elf-objcopy.exe    -v -O binary buttons.elf buttons.gba
copy from buttons.elf(elf32-littlearm) to buttons.gba(binary)
G:/ham/tools/win32/rm -f *.o *.i *.ii
```

Figure 4.14

After compiling the project, the status window at the bottom of Visual HAM displays messages from the compiler.

Testing the ButtonTest Program in the Emulator

One last time, I'll go over the run process, just to make sure you've got it down. VisualBoyAdvance is the GBA emulator that comes with HAM and may be invoked from the IDE by opening the Project menu and selecting Build + VBA, as shown in Figure 4.15, or by simply pressing the F7 key.



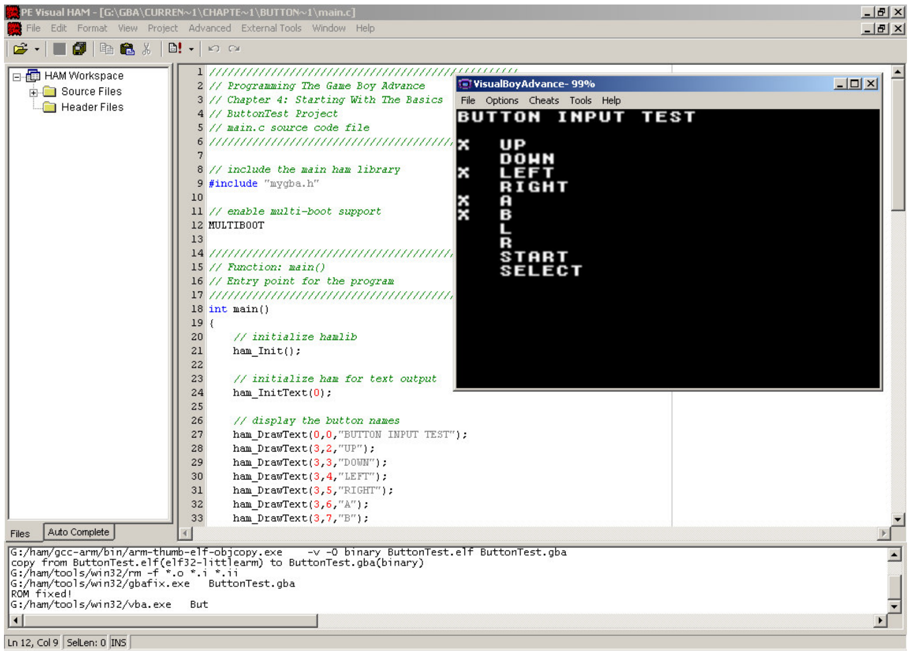
```
//////////////////////////////////////
The Game Boy Advance
tarting With The Basics

ource code file
//////////////////////////////////////

main ham library
.h"

//////////////////////////////////////
in()
for the program
//////////////////////////////////////

    // initialize hamlib
    ham_Init();

    // initialize ham for text output
    ham_InitText(0);

    // display the button names
    ham_DrawText(0,0,"BUTTON INPUT TEST");
    ham_DrawText(3,2,"UP");
    ham_DrawText(3,3,"DOWN");
    ham_DrawText(3,4,"LEFT");
    ham_DrawText(3,5,"RIGHT");
    ham_DrawText(3,6,"A");
    ham_DrawText(3,7,"B");
    ham_DrawText(3,8,"L");
    ham_DrawText(3,9,"R");
    ham_DrawText(3,10,"START");
```

```
G:/ham/gcc-arm/bin/arm-thumb-elf-as.exe    -mthumb-interwork  G:/ham/system/crt0.s -ocrt0.o
G:/ham/gcc-arm/bin/arm-thumb-elf-gcc.exe     -I G:/ham/gcc-arm/include -I G:/ham/gcc-arm/arm-thumb-elf/include -I G:/ham/include -I G:/ham/syst
G:/ham/gcc-arm/bin/arm-thumb-elf-ld.exe     -L G:/ham/gcc-arm/arm-thumb-elf/lib/normal -L G:/ham/gcc-arm/lib -L G:/ham/gcc-arm/lib/gcc-lib/arm-
G:/ham/gcc-arm/bin/arm-thumb-elf-objcopy.exe     -v -O binary buttons.elf buttons.gba
copy from buttons.elf(elf32-littlearm) to buttons.gba(binary)
G:/ham/tools/win32/rm -f *.o *.i *.ii
```

Figure 4.15

The Build + VBA menu item will compile a program and run it in the emulator.

When you do this, Visual HAM will start the compile process. If the program compiles without errors, the program is run in the emulator, which then appears on the screen, as shown in Figure 4.16.


PE Visual HAM - [G:\GBA\CURREN~1\CHAPTE~1\BUTTON~1\main.c]
File Edit Format View Project Advanced External Tools Window Help
HAM Workspace
Source Files
Header Files
Files
Auto Complete
VisualBoyAdvance- 99%
File Options Cheats Tools Help
Ln 12, Col 9
SelLen: 0
INS

```
//////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 4: Starting With The Basics
// ButtonTest Project
// main.c source code file
//////////////////////////////////////////////////

// include the main ham library
#include "mygba.h"

// enable multi-boot support
MULTIBOOT

//////////////////////////////////////////////////
// Function: main()
// Entry point for the program
//////////////////////////////////////////////////
int main()
{
    // initialize hamlib
    ham_Init();

    // initialize ham for text output
    ham_InitText(0);

    // display the button names
    ham_DrawText(0,0,"BUTTON INPUT TEST");
    ham_DrawText(3,2,"UP");
    ham_DrawText(3,3,"DOWN");
    ham_DrawText(3,4,"LEFT");
    ham_DrawText(3,5,"RIGHT");
    ham_DrawText(3,6,"A");
    ham_DrawText(3,7,"B");
```

```
BUTTON INPUT TEST

X   UP
    DOWN
X   LEFT
    RIGHT
X   A
X   B
    L
    R
    START
    SELECT
```

```
G:/ham/gcc-arm/bin/arm-thumb-elf-objcopy.exe     -v -O binary ButtonTest.elf ButtonTest.gba
copy from ButtonTest.elf(elf32-littlearm) to ButtonTest.gba(binary)
G:/ham/tools/win32/rm -f *.o *.i *.ii
G:/ham/tools/win32/gbafix.exe    ButtonTest.gba
ROM fixed!
G:/ham/tools/win32/vba.exe    But
```

Figure 4.16

The ButtonTest program has been compiled and is shown running in the emulator.

Running Programs Directly on the GBA

The Game Boy Advance is a portable video game machine, so sooner or later you will want to take your games and demos with you or at least be able to run the programs on the actual GBA—to see how well it plays on the real hardware. There are two ways to do this. First, there is the Multi Boot Version 2 (MBV2) device, which connects the GBA to your PC using a parallel port adapter. Then there is the Flash Advance Linker (of which there are many different models and types), which can read and write to flash cartridges. If you end up purchasing one of these hardware devices, you will get instructions on how to use it.

Since there are so many devices available, I won't go into detail about specifically what steps to take to install the driver (if there is one) or how to use the software. When you connect either your MBV2 or Flash Advance Linker to your PC and are able to successfully use it as described in the product's enclosed instructions (which may be printed or may be in electronic form on an enclosed floppy disk or CD-ROM), then you will be able to use it with Visual HAM. The process is fairly easy from that point forward. The initial installation and testing require more effort than actually using one of these devices, but the reward—of being able to see your programs running on a real GBA—are definitely worth the cost and effort of acquiring and installing one.

The Multiboot Cable

The multiboot device (formally known as the Multi Boot Version 2, or MBV2) allows you to run programs directly on the GBA without a flash linker by taking advantage of the GBA's multiplayer capabilities. When a GBA detects a multiplayer cable inserted into the Ext port, it will attempt to download a small binary program into memory from the host GBA (which is running the host game—for instance, *Mario Kart Super Circuit*). See Figure 4.17 for a picture of an MBV2 device. Games with multiboot capability allow up to four players to participate in a game where only one of the players is using the actual game cartridge.

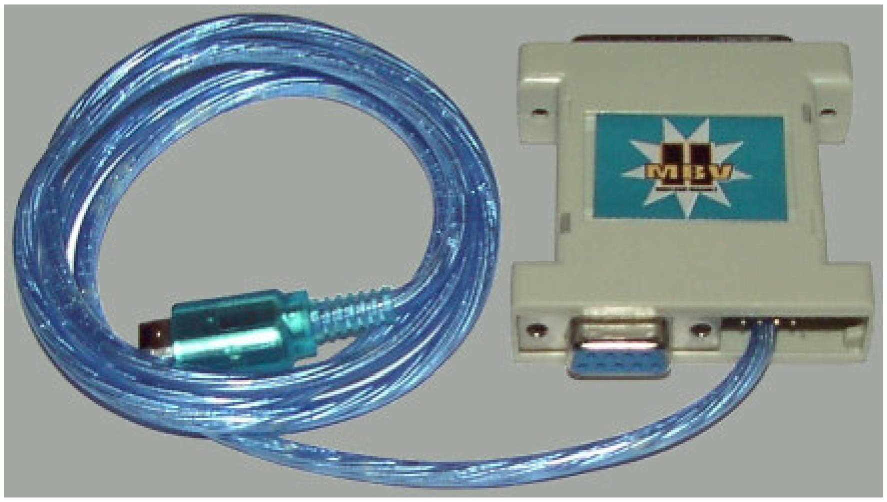
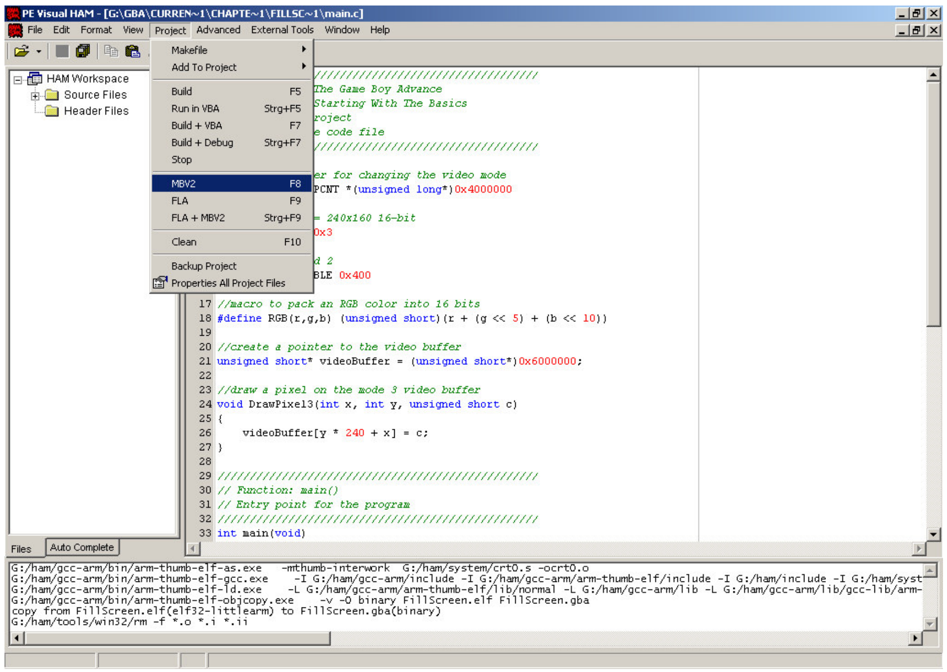


Figure 4.17
The Multi Boot Version 2 is a cable that connects a GBA to a PC using a parallel port adapter.

The great thing about the MBV2 is that it is a low-cost development device that complements Visual HAM wonderfully. HAM even includes built-in support for MBV2, as the transfer software is installed with HAM. In order to use the MBV2, you need to plug it into the parallel (printer) port on your PC, and then connect the blue cable to the link port on your GBA. Remove any cartridge from the GBA, so the cartridge slot is empty. Then turn on the GBA. Now, from within Visual HAM, you can choose to compile and run the program directly on the GBA.

Take a look at Figure 4.18, which shows the Project menu in Visual HAM, with the MBV2 menu item highlighted. Just be sure to compile the program first by pressing F5, and then you can send the program to the MBV2 device by pressing F8 (which causes the program to run on your GBA, so be sure to have your GBA power turned on before starting an MBV2 session).



```
//////////////////////////////////////
The Game Boy Advance
Starting With The Basics
roject
e code file
//////////////////////////////////////

er for changing the video mode
PCNT *(unsigned long*)0x4000000

= 240x160 16-bit
0x3

d 2
BLE 0x400

//macro to pack an RGB color into 16 bits
#define RGB(r,g,b) (unsigned short)(r + (g << 5) + (b << 10))

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

//draw a pixel on the mode 3 video buffer
void DrawPixel3(int x, int y, unsigned short c)
{
    videoBuffer[y * 240 + x] = c;
}

//////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
//////////////////////////////////////////////////////
int main(void)
```

```
G:/ham/gcc-arm/bin/arm-thumb-elf-as.exe   -mthumb-interwork  G:/ham/system/crt0.s -ocrt0.o
G:/ham/gcc-arm/bin/arm-thumb-elf-gcc.exe    -I G:/ham/gcc-arm/include -I G:/ham/gcc-arm/arm-thumb-elf/include -I G:/ham/include -I G:/ham/syst
G:/ham/gcc-arm/bin/arm-thumb-elf-ld.exe    -L G:/ham/gcc-arm/arm-thumb-elf/lib/normal -L G:/ham/gcc-arm/lib -L G:/ham/gcc-arm/lib/gcc-lib/arm-
G:/ham/gcc-arm/bin/arm-thumb-elf-objcopy.exe    -v -O binary FillScreen.elf FillScreen.gba
copy from FillScreen.elf(elf32-littlearm) to FillScreen.gba(binary)
G:/ham/tools/win32/rm -f *.o *.i *.ii
```

Figure 4.18

Start an MBV2 session in Visual HAM by selecting Project, MBV2.

Figure 4.19 shows the complete MBV2 retail kit, which comes with a disk containing the MBV2 driver and transfer software. Although aftermarket development accessories like the MBV2 are not available through retail channels, you may order the MBV2 kit on the Web, from sources such as http://www.lik-sang.com. You may also find other sources for this kit by going to a search engine and typing in "game boy advance multi-boot", which should return a list of sites selling this device (and others like it).

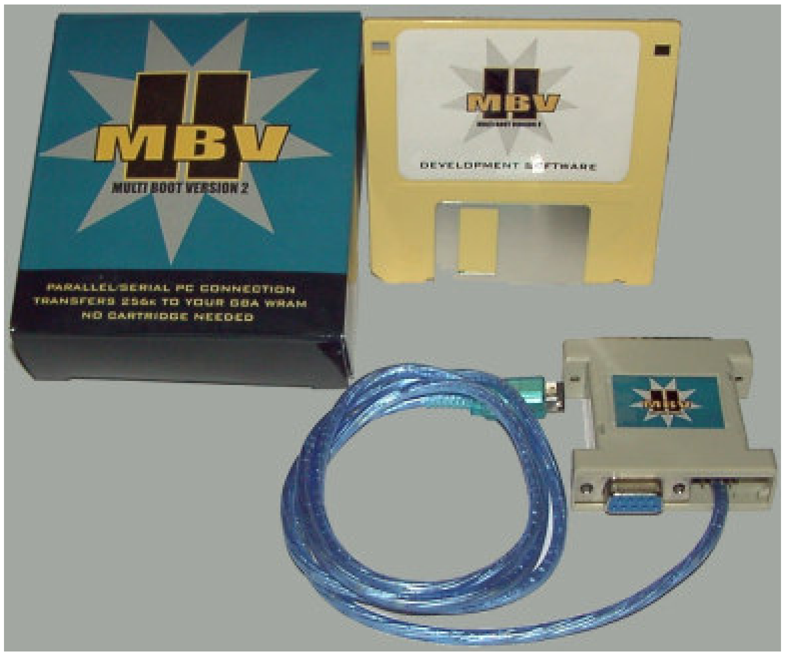


Figure 4.19
The MBV2 kit, showing the included software disk and retail packaging.

The Flash Advance Linker

The Flash Advance Linker (shown in Figure 4.20) is a parallel port device capable of reading and writing GBA cartridges. This device is used to copy your compiled GBA programs to a blank flash cartridge that is compatible with the GBA game cartridge slot. After writing the binary ROM for your program to the cartridge, you then remove it and insert it into your GBA, at which point it functions like any regular game cartridge.

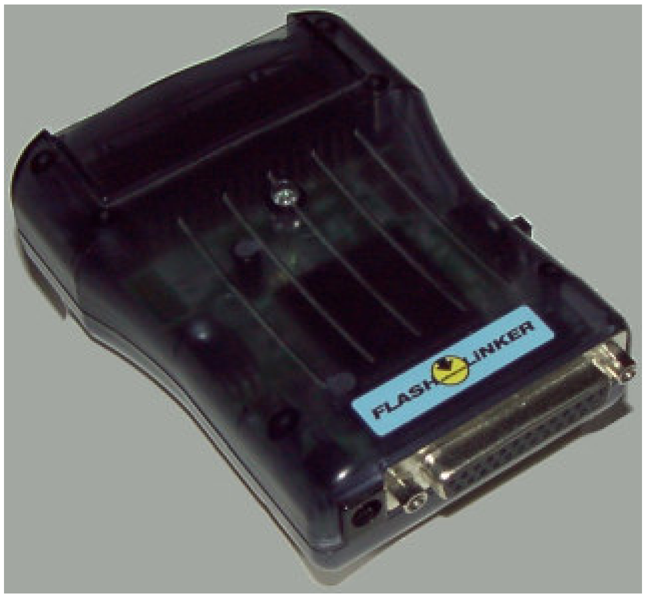


Figure 4.20
The Flash Advance Linker connects to a PC using a parallel port cable.

The flash cartridges used with the Flash Advance Linker come in varying sizes, including 64, 128, 256, 512, and even 1,024 Mbits. The standard 64M cartridge, shown in Figure 4.21, is one of the options available when you purchase your own Flash Advance Linker.

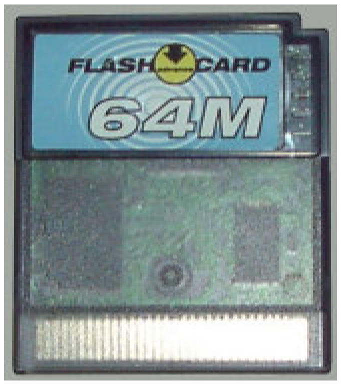


Figure 4.21

A rewritable flash cartridge comes with the Flash Advance Linker and is compatible with the GBA cartridge slot.

Unlike the MBV2, which connects directly to the parallel port and provides a link cable to your GBA, the Flash Advance Linker is a bulkier device, requiring a parallel cable to connect to your PC, and there is no direct connection to a GBA. Figure 4.22 shows the device with a parallel cable connected to it.

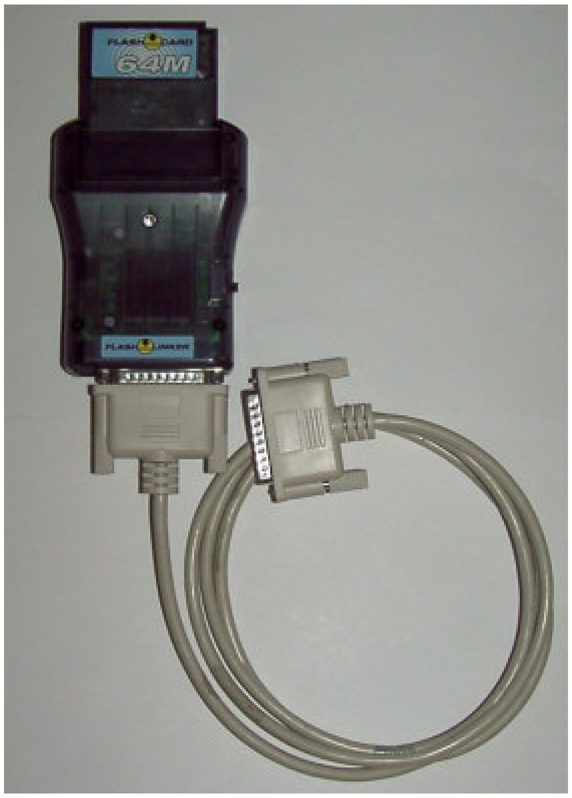


Figure 4.22

The Flash Advance Linker device with a flash cartridge inserted and a parallel cable attached.

Summary

Although the material in this chapter has been introductory in nature, providing the first working code samples for the book, you now have all the tools needed to write a rudimentary game by drawing simple shapes on the screen and detecting button input. In addition to providing a first glimpse into GBA programming, this chapter also provided an overview of the hardware accessories that allow you to develop and run programs directly on a GBA.

Challenges

The following challenges will help to reinforce the material you have learned in this chapter. The solution to each challenge is provided on the CD-ROM inside the folder for this chapter.

Challenge 1: Modify the Greeting program so that it displays a different message on the screen depending on which button has been pressed.

Challenge 2: Modify the DrawPixel program so that it moves the pixel around the screen and causes the pixel to bounce off the edges of the screen.

Challenge 3: Modify the FillScreen program by moving the for loops into a function called FillScreen that fills the screen with a specified RGB color.

Chapter Quiz

The key to the quiz may be found in Appendix D.

1. What language is featured in this book for writing Game Boy Advance programs?
    - A. C++
    - B. Basic
    - C. C
    - D. Prolog

2. What is the Hamlib function for displaying text on the screen?
    - A. ham_DisplayText
    - B. ham_DrawText

C. ham_PrintText
D. ham_SetText

3. What is the name of the GBA emulator used in this book (and distributed with HAM)?
    A. VisualBoyAdvance
    B. Visual Game Emulator
    C. GBA-EMU
    D. WinGBA

4. True or False: The HAM distribution comes with flash linker and multiboot software.
    A. True
    B. False

5. What is the display resolution of video mode 3?
    A. 320 x 240
    B. 260 x 180
    C. 120 x 80
    D. 240 x 160

6. What is the name of the software development kit *distribution* used in this book?
    A. Visual HAM
    B. HAM
    C. Hamlib
    D. DevKit-Advance

7. What is the color depth of the screen in video mode 3?
    A. 8 bits
    B. 16 bits
    C. 24 bits
    D. 32 bits

8. What is the name of the memory register used to change the video mode?
    A. REG_CHGMOD
    B. REG_MODECH
    C. REG_DISPCNT
    D. REG_DMA01

9. What is the name of the Hamlib function that initializes the text display system?
    A. ham_StartText
    B. ham_LoadText
    C. ham_BeginText
    D. ham_InitText

10. What parallel port device connects to the Ext port on the GBA in order to run programs directly on the GBA?
    A. Multi Boot Version 2 (MBV2)
    B. VisualBoyAdvance (VBA)
    C. Flash Advance Linker (FLA)
    D. Major League Baseball (MLB)