Chapter 11

ARM7 Assembly Language Primer

This chapter is a brief overview of the ARM7 instruction set and assembly-language primer for the GBA. Assembly language is not difficult to understand, but mastering the subject requires time and patience. This chapter provides enough information for you to write a complete assembly-language program from scratch and also shows how to write assembly functions and call them from a main C program, which is most likely something that you will be doing sooner or later as you write complete GBA games.

For starters, you will learn how to compile a program from the command prompt using the HAM compiler chain manually, a necessary first step before assembling and linking the programs in the chapter. This is not a comprehensive chapter on ARM7 assembly language, by any means. In fact, it is rather sparse on the instruction set. The goal of this chapter is not to teach you how to write assembly language, but rather, how to use assembly to enhance a C program, with a few simple examples. Please refer to the reference books listed later in this chapter (as well as in Appendix B) that cover the ARM architecture.

Here is a rundown of the subjects in this chapter:

- Introduction to command-line compiling
- Basic ARM7 assembly language
- Calling assembly functions from C

Introduction to Command-Line Compiling

Before getting into assembly language, I would like to first explain how to compile a regular C program from the command line, because you will need this information in order to use the command-line tools. Visual HAM and the HAM SDK hide away the compiler chain quite well, which is primarily why I chose these tools for the book. Now that you have made it through the thick and thin of GBA coding and are ready for a little "pedal to the metal," I need to show you how to compile programs completely outside of Visual HAM--without even the benefit of a "make" file. A make file is a text file that describes the process of compiling, assembling, and linking source code files into a final binary .gba file, and this is what Visual HAM does when you press F5 or F7 to build the project, it invokes the make utility.

Have you noticed that new projects always come with a file called *"makefile"* with no extension? This is the default file that the make utility loads when you simply type "make" on the command line, with no options. It loads the default file and processes it. The make file has pathnames and specifies what to include in the compile options to take a simple C source file and produce a GBA ROM image out of it--which is no easy matter. It's just that HAM makes this very easy because it includes the complete compiler chain (all the utilities, assemblers, compilers, includes, and libs needed to build a GBA ROM).

Now what I'd like to do is take one of the sample programs from an earlier chapter and show you how to compile it manually. You'll then write a few short batch files that can be reused to compile other programs (including assembly-language files) into a .gba image.

Compiling from the Command Line

The TestBuild project on the CD-ROM was adapted from the simple DrawPixel program covered previously. This project is located in \Sources\Chapter11\TestBuild. Remember, you don't need to load this into Visual HAM, because this will be a command-line exercise! It's a very short program (without the comment lines), so I'll list the main.c file here. Use Notepad or a similar text editor to type in the program.

```
//////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 11: ARM7 Assembly Language Primer
// TestBuild Project
// main.c source code file
//////////////////////////////////////////////////

int main(void)
```

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

    //enter video mode 3
    *(unsigned long*)0x4000000 = (0x3 | 0x400);

    //draw a white pixel centered on the screen
    videoBuffer[80 * 240 + 120] = 0xFFFF;

    while(1);
    return 0;
}
```

Creating a Compile Batch File

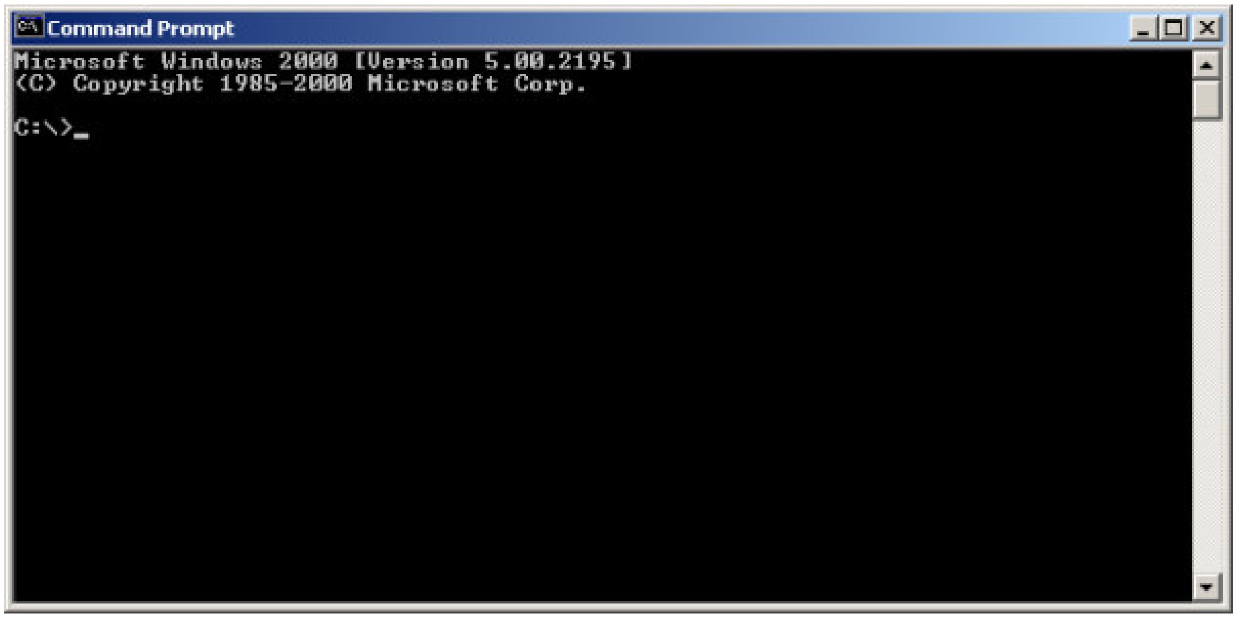
Compiling the program will require more work than typing in the program itself, unfortunately! But I'm going to make it so you only have to type in the compile options once in a batch file, which you can then reuse for the remaining programs in this chapter.

So, go ahead and open up Notepad again, type in the following, and then save the file as gcc.bat. I'll explain what it does after you have tried it out first. Although it takes up several lines in the listing, this is actually just one long line. If you have word wrap enabled in Notepad (via the Format menu), then just type in the compile command without any line breaks. Here is the entire command:

```
arm-thumb-elf-gcc.exe -I %HAMDIR%\gcc-arm\include -I %HAMDIR%\include
-I %HAMDIR%\gcc-arm\arm-thumb-elf\include -I %HAMDIR%\system -c -DHAM_HAM
-DHAM_MULTIBOOT -DHAM_ENABLE_MBV2LIB -O2 -DHAM_WITH_LIBHAM
-mthumb-interwork -mlong-calls -Wall -save-temps -fverbose-asm
%1.c -o%1.o
```

Don't forget the last short line--it's the most important one! Believe it or not, it takes that huge command just to compile a single C source file with GCC, due to all the include files and compiler directives that have been set up for GBA development.

Now let's try out the command. Bring up the Start menu in Windows, and select Programs, Accessories, Command Prompt. It should look like Figure 11.1.



```
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\>_
```

Figure 11.1 - The Command Prompt as it appears in a stock Windows 2000 system.

Now, my GBA development tools and sources are all on another hard drive partition (G), so I'm going to switch over to that drive by typing "G:" and pressing Enter. I'll explain some of these steps because not everyone has a lot of experience with the command prompt (although I imagine some may recall the old MS-DOS days?). I've created a folder called TestBuild inside \GBA\Sources\Chapter11. If you have just copied the \Sources folder off the CD-ROM entirely, then ignore my specific folder names and follow your own configuration. I'm going to type

```
CD \GBA\Sources\Chapter11\TestBuild
```

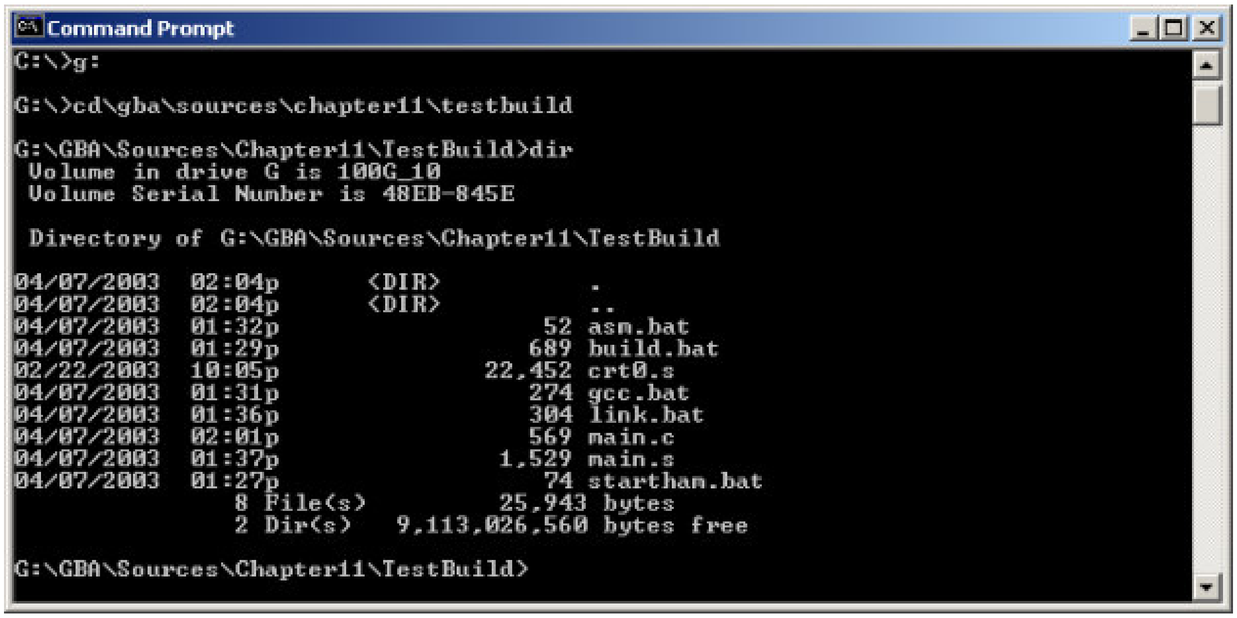
to get into the correct folder for the TestBuild project. A list of files in this folder is shown in Figure 11.2, just to be sure we're in sync. If you have copied the sources off the CD-ROM to your hard drive, then you will want to replace the pathname here with the correct path to chapter 11 on your system.

Creating a Path to the Compiler Chain

Notice that there are some batch files in there other than the gcc.bat file you just created. One such file is startham.bat. This file is generated by HAM during installation and is located in \HAM. I have copied the file to the TestBuild folder so it's easier to use and edited the comments out of the file. Here is what it looks like:

```
set PATH=g:\ham\gcc-arm\bin;g:\ham\tools\win32;%PATH%

set HAMDIR=g:\ham
```



```
C:\>g:

G:\>cd\gba\sources\chapter11\testbuild

G:\GBA\Sources\Chapter11\TestBuild>dir
 Volume in drive G is 100G_10
 Volume Serial Number is 48EB-845E

 Directory of G:\GBA\Sources\Chapter11\TestBuild

04/07/2003  02:04p      <DIR>          .
04/07/2003  02:04p      <DIR>          ..
04/07/2003  01:32p                  52 asm.bat
04/07/2003  01:29p                 689 build.bat
02/22/2003  10:05p              22,452 crt0.s
04/07/2003  01:31p                 274 gcc.bat
04/07/2003  01:36p                 304 link.bat
04/07/2003  02:01p                 569 main.c
04/07/2003  01:37p               1,529 main.s
04/07/2003  01:27p                  74 startham.bat
               8 File(s)         25,943 bytes
               2 Dir(s)   9,113,026,560 bytes free

G:\GBA\Sources\Chapter11\TestBuild>
```

Figure 11.2 - The list of files inside the TestBuild project folder.

This batch file opens up a path to the HAM compiler chain and tools folders so you can invoke those tools from anywhere. The paths in this batch file should reflect your installation of HAM; if it differs, then be sure to correct the paths before running the batch file. Type **startham** now to set up the command-line paths.

Compiling The Program

You are now ready to compile the main.c file using gcc.bat, which you created a moment ago. Here is how you compile the main.c file:

```
gcc main
```

Notice that I didn't include the .c extension. That's because the gcc.bat file adds the extension automatically. It needs the file name without the extension because it uses that name to generate the output file, which will be main.o. In the command-line realm, no news is good news. If no messages are printed on the command line, that means the GCC compiler compiled the program without any warnings or errors. Now, there will be a command displayed on the screen when you type "gcc main", because the batch file echoes the command on the screen. If you prefer to hide it, you can add **@echo off** to the top of the batch file, before the command, and that will hide the commands.

You can look in the TestBuild folder after running the command, and you should see a new file, main.o. This is the object file, containing machine instructions for the ARM7 chip in the GBA. These instructions are specific only to the program and do not know how to boot up or anything like that. There are a couple more steps involved before the main.o file can be converted to a .gba file (a process called linking the object files). I will show you how to link the main.o file shortly. In the meantime, take a look at Figure 11.3, which shows an illustration of the compile process.

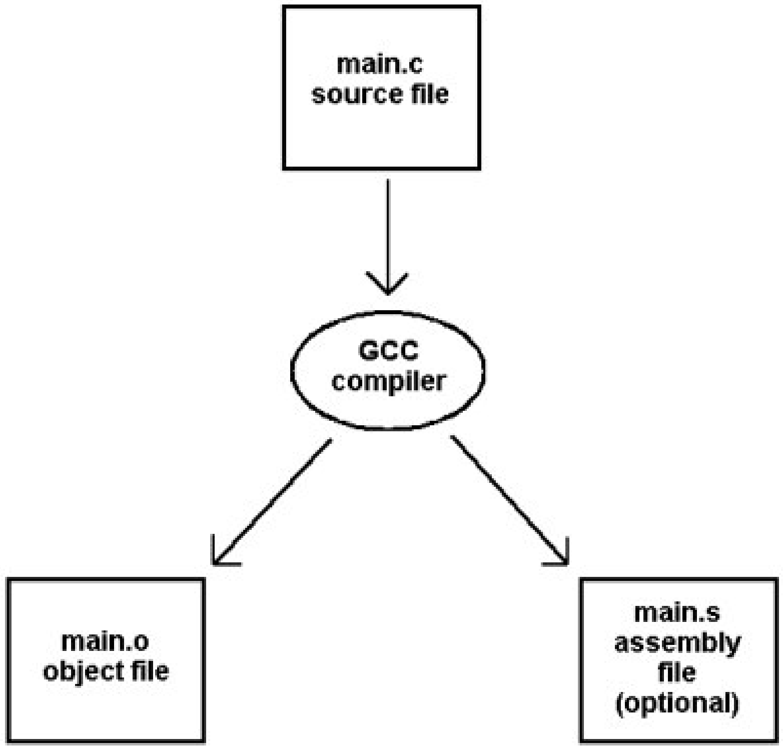


Figure 11.3
Compiling a source code file into an object file with the GCC compiler.

Assembling from the Command Line

In the GBA development realm, object files have an extension of .o. That is why the main.c file was compiled to main.o when you invoked **gcc main** a moment ago. The linker is another tool required to build a final .gba file, as it takes all the various object files and links them together into a single object file, which can then be turned into a .gba (using another command-line program that I will show you shortly). Before I give you the link command, though, one small step must first be done.

There is an assembler that comes with HAM and is located in the same place as the compiler. The exact file name of the assembler is **arm-thumb-elf-as.exe**, where the "as" means "assembler." Assembly language is the lowest programming language as far as being close to the hardware. Indeed, assembly is very difficult to master and is not for the faint of heart, because each assembly instruction translates directly to a CPU instruction! So you are literally working directly with the innards of the CPU. Since this is a practical chapter rather than a theoretical one, I'm going to show you how to assemble a file without really explaining what is in that file. The file is called crt0.s, and it contains all the bootstrap and execution code needed to build a GBA ROM image. This crt0.s file must be assembled and linked together with your main.o object file in order to run it on a GBA (or inside an emulator).

The crt0.s file is a bootstrap source code file that provides all the services needed by a GBA program, such as GBA initialization. The crt0.s assembly file is what you might call the boot sector program. On a PC, there is a boot sector on each hard drive, and the very first sector on the hard drive includes operating system bootup instructions. The BIOS (basic input-output system) on the PC, after checking over the hardware on the PC, will invoke this small program on whatever drive is available. Usually, the first drive is a floppy drive, which will try to boot if you insert a disk when powering up the PC. On most PCs, the CD-ROM is also bootable, although it is the hard drive that boots most of the time. The operating system installs a boot sector program that is run by the BIOS, and this boot sector program will run an operating system loader. For instance, the operating system loader for Windows is command.com, and has been that same filename since the very first version of MS-DOS. The crt0.s program is similar to the bootstrap program on a PC, only it is designed to boot up the GBA. Remember, the ROM image in your .gba program is all the GBA has to go by, as there is no operating system on the GBA, so the bootstrap must be included in the ROM!

I have copied the crt0.s assembly file out of the HAM folder and placed it inside the TestBuild project folder, because I want to eliminate long pathnames as much as possible. To assemble the file, you will want to type in the following command:

```
arm-thumb-elf-as.exe -mthumb-interwork crt0.s -ocrt0.o
```

The reason why this file name is so long is because it's descriptive. The ARM7 processor has two different modes built in, and you may switch between them at any time. The modes are ARM and THUMB. ARM is the full 32-bit instruction set, while THUMB is a 16-bit hardware-emulated instruction set. While the programs in this chapter do no use any thumb code, I want to include support for this mode because you may want to reuse the commands stored in the gcc.bat batch file (and the other batch files in this chapter).

Creating an Assemble Batch File

As usual, there will be no message if the file was assembled correctly without error, and there will be a new file called crt0.o in the TestBuild folder, so go ahead and take a look. You now have the boot object file and your main object file and are ready to link. But first I want to turn that assembly command into a more convenient batch file that can be reused (as it will be later).

Create a new text file called asm.bat and type the following line into this file:

```
arm-thumb-elf-as.exe -mthumb-interwork %1.s -o%1.o
```

This is the same command, essentially, but the file name has been replaced with a parameter, %1, that will fill in the file name passed to it. Now, using this batch file, you can assemble any .s file, (for instance: **asm crt0**), noting again the lack of an extension, as it is automatically filled in. Figure 11.4 illustrates how the assembler works.

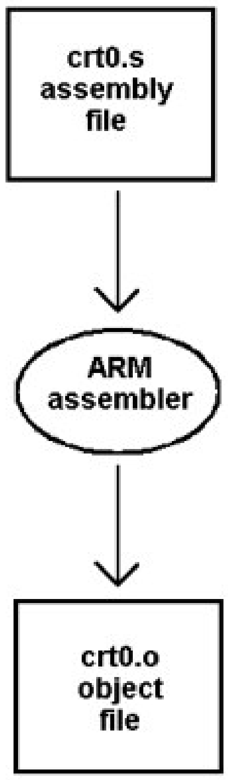


Figure 11.4
Assembling an assembly language file into an object file with the ARM assembler.

Linking from the Command Line

The two object files have been created--one from the main.c source file, the other from the crt0.s assembly-language file--and are ready to be linked together. The linking process is more involved, because a lot of libraries must be included in order to satisfy all the function calls within crt0.o and main.o. I'm not going to spend much time explaining all the libraries because they are just part of the GCC compiler chain for the GBA.

Creating a Linker Batch File

Okay, let's get started, this time with the batch file right away. Because I only want to go over the command itself once, we might as well just stick it into a batch file. Create a new file and call it link.bat, typing the following command into the file. Again, if you're using Notepad, simply type away and don't fill in any new lines, as this should be a single long command. After typing in the link command, there is one more command to be added to this batch file, a call to objcopy that actually takes the linked .elf file and converts it to a .gba file. So these two commands are both inside the link.bat file.

```
@echo off
arm-thumb-elf-ld.exe -L %HAMDIR%\gcc-arm\lib\gcc-lib\arm-thumb-elf\3.2.2\normal
-L %HAMDIR%\gcc-arm\arm-thumb-elf\lib\normal -L %HAMDIR%\gcc-arm\lib
--script %HAMDIR%\system\lnkscript-afm -o%1.elf %1.o crt0.o -lafm -lham -lm
-lstdc++ -lsupc++ -lc -lgcc

arm-thumb-elf-objcopy.exe -v -O binary %1.elf %1.gba
```

Make sure these two commands are both inside the link.bat file, and typed without using the Enter key, and then you can link the program with the following command:

```
link main
```

If all goes well, you should see the commands echoed to the screen, but otherwise no error messages. Which means. . .you now have compiled your first GBA program the hard way! The program should look like Figure 11.5. If the link process worked without error, you should see a main.gba file, which you can load into the emulator.

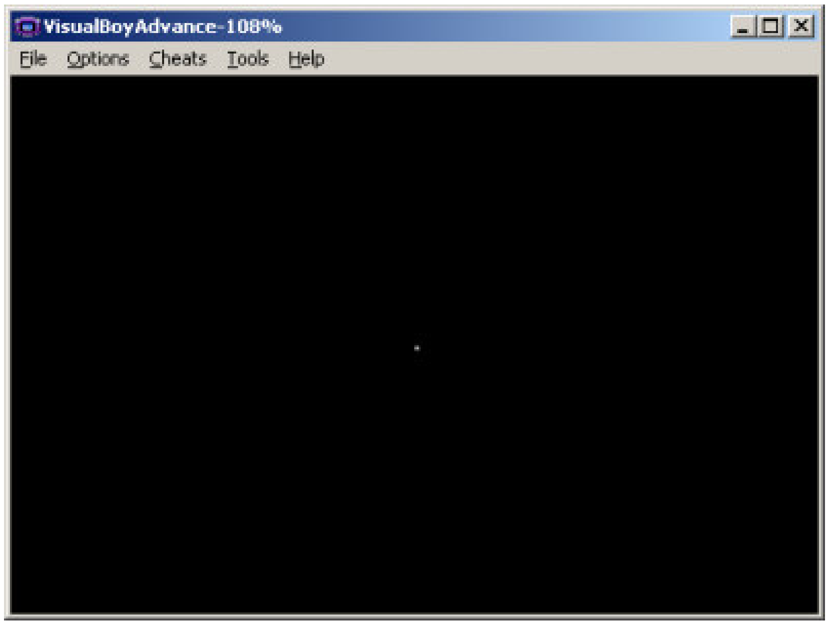


Figure 11.5
The TestBuild program draws a pixel as expected.

If instead you get an error message related to an unrecognized command, the problem is most likely due to a linebreak in the batch file. Make sure that each of the two commands are free of linebreaks. You may also double check to ensure that you ran startham.bat first, in order to set up the environment variables for the command-line tools.

Keep the batch files you have created--asm.bat, gcc.bat, and link.bat--handy, as you'll need them for the next program. It's incredibly beneficial to understand how the compiler, assembler, and linker works, so this experience will be valuable to you in your GBA coding efforts. Don't rely solely on Visual HAM and the HAM SDK to do all the work for you. After all, HAM was originally designed to be run from the command line in order to build GBA programs. Visual HAM came later. Figure 11.6 illustrates the process of linking object files into an executable file.

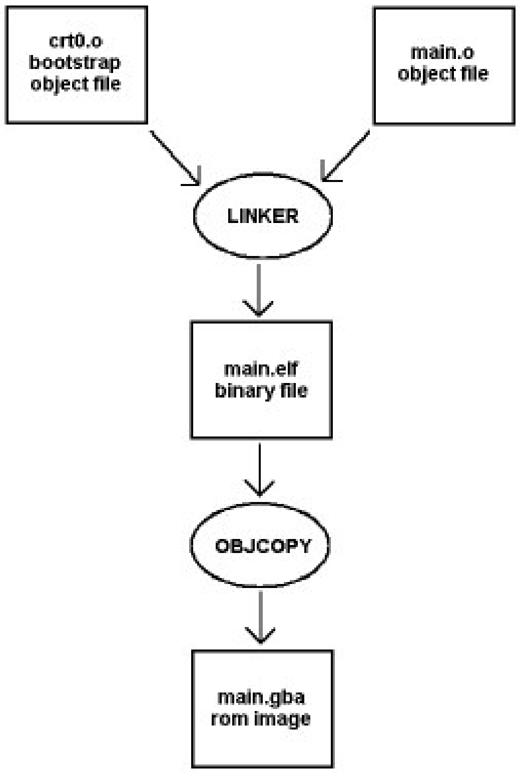


Figure 11.6

Converting object files into an executable rom image file.

(Very) Basic ARM7 Assembly Language

In this section I'm going to walk you through two very simple assembly-language programs help you gain a little appreciation for the C language, for one thing, but also to familiarize you with what an assembly file looks like and how you can start to enhance your GBA programs with very low-level code. Unfortunately, ARM7 assembly language is a complicated subject, and I am only showing you *what* it is in this chapter, rather than going into any detail about *how* to use it to the fullest extent. For details on the ARM7TDMI CPU and its instruction set, you may refer to an online reference (go to www.google.com and search for "arm7tdmi"), or you may order a reference book (see Appendix B).

There is an excellent tutorial on ARM7 assembly language on the Web at http://k2pts.home.attbi.com/gbaguy, and another great online reference guide at http://re-eject.gbadev.org/. Just keep in mind that nothing on the Web is permanent, and these URLs are subject to change.

Assembly instructions are very similar to the CPU instructions, which is why you'll see mov, add, str, b, and other obscure statements, often combined with one or more registers. That's also why assembly is such a difficult language to master. The ARM7 chip has many general-purpose registers, such as r1, r2, r3, r4, r5, and so on. While it's true that once you have learned assembly for one processor, you have a good chance of quickly learning assembly language for other processors, the architectures can vary widely. For instance, the ARM7 is a reduced instruction set computer (RISC) chip, while common PC processors from Intel, AMD, and others are complex instruction set computer (CISC) chips.

The difference in these architectures is significant but may be broken down into two main areas: registers and instructions. A RISC chip has many registers but few instructions. A CISC chip has only a few registers but many instructions. The supposed extra performance in a RISC chip is due in part to the long pipelines in many processors today. The Intel Pentium 4 processor has a 20-stage pipeline. If this were a RISC chip, it would be much faster, because branch prediction would be more accurate (due to the many available registers and fewer instructions). The ARM7 has a 3-stage pipeline, which is very good for a low-voltage and small footprint processor (that runs on two AA batteries!). These are all subjects relegated to a hardware discussion, and I mention them simply to make a point, so don't worry if you are unfamiliar with hardware terminology, it's not important for writing GBA code.

A Simple Assembly Program

The really practical aspect of this chapter is that I want to show you not a list of the ARM7 instruction set, but rather only a few useful instructions that can be used inside a given function to enhance a C program. For starters, I'll show you how to write your first assembly-language program that will run on the GBA. This program is called FirstAsm and is shown running in Figure 11.7. Not very impressive, but it's accessing the video buffer and drawing a pixel, all in 100 percent assembly language. Are you intrigued? It's a *lot* of fun writing your first assembly program! Here we go.

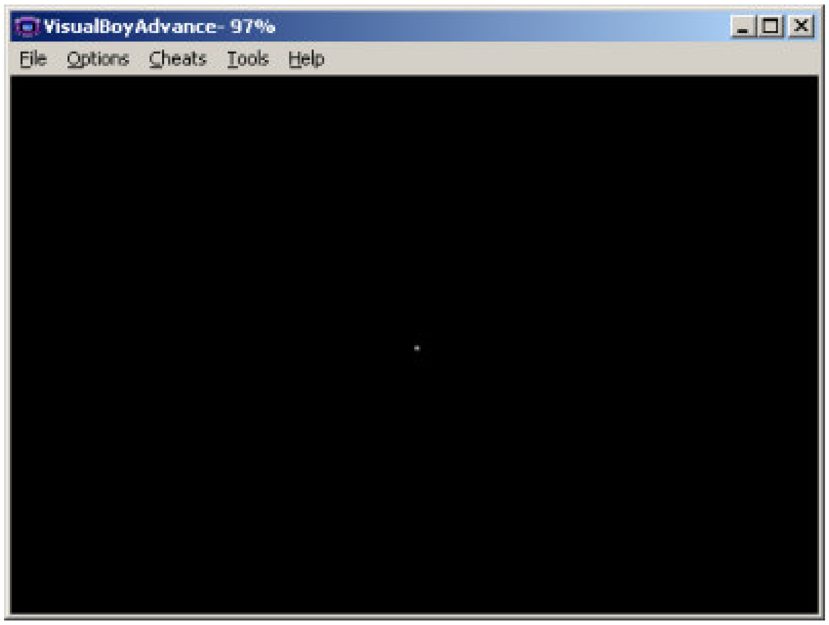


Figure 11.7

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

The FirstAsm Program

The FirstAsm program is a simple assembly listing that sets the video mode to mode 3 with background 2 enabled and then draws a pixel at the center of the screen. I have included comments with each line. As you can see from the listing that follows, a comment in ARM7 assembly starts with the @ character. Anything that falls on the same line after that character is ignored by the assembler. Now, you are going to want to type this code into a file called pixel.s.

```
@////////////////////////////////////////////////////
@ Programming The Game Boy Advance
@ Chapter 11: ARM7 Assembly Language Primer
@ FirstAsm Program
@ pixel.s assembly file
@////////////////////////////////////////////////////

    .text
    .align2
    .globalmain

@main entry point of program
main:
```

```
@set video mode 3 using background 2
    mov    r2, #1024            @BG2_ENABLE
    add    r2, r2, #3           @mode 3
    mov    r3, #67108864        @REG_DISPCNT
    str    r2, [r3, #0]         @set value

@draw a white pixel at 80x120
@remember, mode 3 is 2 bytes/pixel
    mov    r1, #38400@80*240*2
    add    r1, r1, #240@X=120
    add    r3, r3, #33554432@videoMemory
    mvn    r2, #0  @draw pixel
    strh    r2, [r3, r1]

@endless loop
.forever:
    b   .forever

@define object size of main
.end:
    .sizemain,.end-main
```

All done? Great! Now let's assemble and run this baby. If you created the batch files that I covered earlier, then you should have an asm.bat file available. You may want to copy the asm.bat file and pixel.s files into a new folder that is just for this project. Or feel free to just leave this file with the other files from this chapter, in a single folder. That way you can just use the batch files, and crt0 file will be readily available. On the CD-ROM, these files are all in separate project folders; if you want to just copy them off the CD, feel free to do so. Now, from the command prompt, assuming you are in the correct folder, type this:

```
asm pixel
```

to assemble the source code file for the program. If there were no errors, then you can link the file and run it. I'm assuming you already assembled the crt0.s file earlier in this chapter. If not, refer to the start of the chapter for a tutorial on assembling this file and why it is needed. Now let's link:

```
link pixel
```

That's all there is to it! You should now see a pixel.gba file in the folder. Go ahead and run it in VisualBoyAdvance, just as you have done for all the previous projects in the book. Congratulations! You have just written your first ARM7 assembly-language program.

Calling Assembly Functions from C

The real goal of this chapter is to show you how you can use assembly functions to optimize your GBA games, presumably written in C. To do that, you will need to write the assembly code in a separate .s file and then define an external function prototype in the C source code file. For this example program, which I have called ExternAsm, I will show you how to write a simple DrawPixel function in assembly and then use it from a C program. To keep things simpler on the asm side, I have passed the video buffer to the asm function as a parameter. The function prototype looks like this:

```
extern void DrawPixel32 (u32 x, u32 y, u32 color, u32 videobuffer);
```

Can you imagine the power this gives you? If you learn a little ARM7 assembly language, you can optimize any part of your code that is written in C, making use of the astounding speed of assembly language. Of course, you could even write the entire game in assembly, although I wouldn't recommend it. I knew someone who wrote a Sega Genesis game entirely in 68000 assembly, and it was not a fun experience for him. For one thing, simple things like loading data from a file is an enormous chore in assembly, and the resulting benefits for system-level functionality like that are more sensibly utilized in C, reserving assembly for optimization. That is the method of most game programmers--and it is what I recommend.

Making the Function Call

The actual function prototype is all you need to declare an external function in C. Isn't that easy? Imagine just replacing slow C functions in your game with assembly by simply adding an external function prototype! Here's the source code for the extern.c file, the main C source code file for the ExternAsm program. I will go over the actual DrawPixel32 assembly function next.

```
//////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 11: ARM7 Assembly Language Primer
// ExternAsm Project
// extern.c source code file
//////////////////////////////////////////////////

typedef unsigned long u32;
```

```
//declare prototype for the external assembly function
extern void DrawPixel32 (u32 x, u32 y, u32 color, u32 videobuffer);

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

int main(void)
{
    u32 x, y;

    //set video mode 3
    REG_DISPCNT = (3 | 0x400);

    //fill screen with a pattern
    for (y = 0; y < 159; y++)
        for (x = 0; x < 239; x++)
            DrawPixel32(x, y, x*y%31, 0x6000000);

    while(1);
    return 0;
}
```

The DrawPixel32 Assembly Code

Now for the DrawPixel32 source code. This function was written in ARM7 assembly language, and as you can see, it is very short. Since all four parameters are unsigned int (u32) data types, it was a simple matter to use the registers directly without any intervention code (for instance, to move an 8- or 16-bit number into a 32-bit register, and vice versa). Create a new text file called drawpixel.s and type this code listing into the file, then save it.

```
@ Draw pixel in GBA graphics mode 3
@ DrawPixel32(u32 x, u32 y, u32 color, u32 videobuffer);
@ r0 = x
@ r1 = y
@ r2 = color
@ r3 = videobuffer
```

```
        .ARM
        .ALIGN
        .GLOBL  DrawPixel32
DrawPixel32:
        stmfd   sp!,{r4-r5}      @ Save register r4 and r5 on stack
        mov     r4,#480          @ r4 = 480
        mul     r5,r4,r1         @ r5 = r4 * y
        add     r5,r5,r0,lsl #1  @ r5 = r5 + (x << 1)
        add     r4,r5,r3         @ r4 = r5 + videobuffer
        strh    r2,[r4]          @ *(unsigned short *)r4 = color
        ldmfd   sp!,{r4-r5}      @ Restore registers r4 and r5
        bx      lr
```

Compiling the ExternAsm Program

To compile the ExternAsm program, you will need to compile the extern.c, and assemble the drawpixel.s file, and then link them both. The link.bat file can't accommodate two object files, so I have modified it to accept two object files (by simply adding %2.o to the command;, see the link2.bat file in the ExternAsm folder). If you ever write a program with numerous object files (*.o), then you may need to modify the link.bat file again to accommodate as many .o files as you need. There are ways to add conditional code to a batch file to accommodate as many parameters as are passed to it, but the batch code is somewhat involved, and I don't want to get into it at this point, when this batch file in particular needs only two parameters.

To compile the extern.c file:

```
gcc extern
```

To assemble the drawpixel.s file:

```
asm drawpixel
```

And then, to link them together into a runnable .gba file:

```
link2 extern drawpixel
```

If the compiler, assembler, and linker all returned with no errors, then you have successfully built your first assembly-enhanced program! Quick, run the program in VisualBoyAdvance to see what it looks like. The program's output is shown in Figure 11.8.

Now that's more like it! This program does a lot more than the simple pixel plotter from the last two programs! This program actually fills in the screen with an attractive pattern that is generated entirely using the x and y variables, resulting in what I like to call the red rose effect. Kind of strange, huh? Well, it just goes to show that weird things can happen when you're having fun.

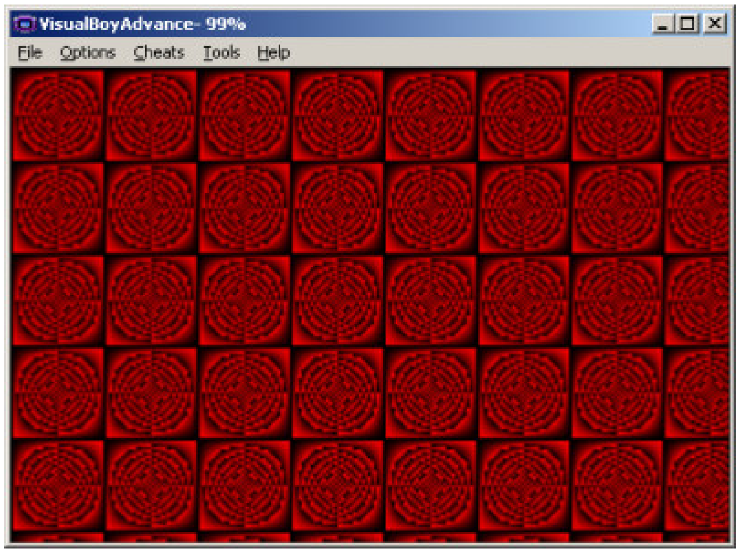


Figure 11.8

The ExternAsm program demonstrates how to call an external assembly function from a C program.

Summary

Well, this has been a fun chapter, but I must admit that there was a lot more that I wanted to cover. The one thing you *don't* want to hear is "Yeah, yeah, it's *beyond the scope of the chapter*." But honestly, that is the truth. I encourage you to learn more about the ARM7 chip that powers the GBA and to learn the instruction set.

To master ARM7 assembly language is to master the GBA, no doubt about it. You will also likely find yourself in gainful employment with a GBA developer, because skilled C programmers who are knowledgeable of the low-level assembly language on a given platform are in high demand.

For starters, I recommend you write a complete game, and then look for ways to optimize it. Look first to your C code, and make sure it is as tight as possible. Then look for bottlenecks that can be improved with assembly. You would be surprised by how even the simplest function implemented in assembly can have a drastic impact on the performance of a game. Good luck!

Challenges

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

Challenge 1: The FirstAsm program draws a pixel in the very center of the screen. See if you can modify the assembly code to have it draw the pixel somewhere else on the screen.

Challenge 2: The ExternAsm program demonstrates how to call an external assembly function. See if you can modify the DrawPixel32 function so that it moves through video memory (which is 38,400 bytes long) and fills the entire screen with a pixel. You may then rename the function to FillScreen32.

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 Appendix D.

1. What extension does an assembly-language file have?

    A. .S

    B. .ASM

    C. .AL

    D. .C

2. What is the 16-bit instruction set on the ARM7 chip called?

    A. ARM

    B. THUMB

    C. HAND

    D. FINGER

3. What is the 32-bit instruction set on the ARM7 chip called?

    A. ARM

    B. THUMB

    C. HAND

    D. FINGER

4. What is the full name of the GCC compiler?

   A. thumb-arm-elf-gcc.exe

   B. if-then-else-gcc.exe

   C. arm-thumb-elf-gcc.exe

   D. open-source-gcc.exe

5. What is the full name of the ARM7 assembler?

   A. hand-finger-thumb-as.exe

   B. hobbit-wizard-orc-as.exe

   C. leg-foot-toe-as.exe

   D. arm-thumb-elf-as.exe

6. What is the extension of the file generated by the linker?

   A. .HBT

   B. .ELF

   C. .ORC

   D. .HMN

7. What does the arm-thumb-elf-objcopy.exe program do?

   A. It links the bitmap and sound files into the main program object file.

   B. It copies an individual object out of a .ELF file into a C array.

   C. It links the object files together into a single .ELF file.

   D. It converts the .ELF object file into a .GBA ROM image file.

8. What does RISC stand for?

   A. Reduced Instruction Set Computer

   B. Really Ignorant Stupid Computer

   C. Radically Integrated Super Computer

   D. Recognition in Some Company

9. How are variables passed from a C calling function to an assembly function?

A. Interrupts

B. Registers

C. Stack

D. DMA

10. True or False: The ARM7 processor is a CISC chip.

A. True

B. False

Epilogue

This book has been, without a doubt, the most enjoyable book I have written so far. Getting down to the bare metal of a console has been an absolute blast, and I am grateful to have been blessed with the opportunity to write this book. I hope you have enjoyed it too! While this has not been a comprehensive reference of the Game Boy Advance, by any means, I believe this book succeeds in the goal I set out for it at the start--to teach anyone of any experience level how to write their own games for the Game Boy Advance. What an experience it has been!

Although I do not know you personally, I have gotten to know many readers and fans of my other books through online forums, so there is a certain feeling of coming full circle at this point. I hope you have found this book not just helpful, but invaluable as a reference, and enjoyable to read. I have strived to cover all the bases of this subject within the context of the goals for this book, and hope you have enjoyed it. There is much more to be learned, and the Game Boy Advance is capable of much, much, much more than what I have presented here! That small ARM7 processor is powerful and can handle fully textured 3D rendering, although that requires a software implementation. I encourage you to seek out the many excellent sample programs and demos written by fans online.

Although every effort was made to ensure that the content and source code presented in this book is error-free, it is possible that errors in print or bugs in the sample programs might have missed scrutiny. If you have any problems with the source code, sample programs, or general theory in this book, please let me know! You can contact me at

support@jharbour.com

and I'll do my best to help you work though any problems. I also welcome constructive criticism and comments that you might have regarding the book in general, or a specific aspect of the book. I get several hundred e-mails a month from readers and respond to every one!

Finally, whether you are an absolute beginner or a seasoned professional, I welcome you to join the discussion list on YahooGroups, where you will have an opportunity to share your games, ideas, and questions with other Game Boy Advance fans! Membership is free and open to the public. Just send an e-mail to the list server at

hamdev@yahoogroups.com

or visit the web site at http://www.yahoogroups.com, and search for the list by name. The list is maintained by Emanuel Schleussinger, the person who created HAM.

Of course, I also recommend you visit my Web site at

http://www.jharbour.com

to keep up to date on the Game Boy Advance development community and any changes or bug reports for the code presented in this book. As always, I look forward to hearing from you!