Part III

Meditating On The Hardware

Welcome to Part III of *Programming The Nintendo Game Boy Advance: The Unofficial Guide*. Part III includes four chapters that are focused on low-level hardware programming, including interfacing with the Game Boy's buttons, using timers and synchronizing objects on the screen, programming the sound system, and interfacing with assembly language.A whole chapter is dedicated to ARM7 assembler, which is the lowest level possible, right down to the bare metal of the Game Boy Advance.

Chapter 8

Using Interrupts And Timers

Up to this point you have learned most of the concepts, theory, and code needed to write entire games for the GBA. Although each chapter in this book largely stands independently of the others, it would have been helpful to read them in order. The flip side to delving into the GBA's graphics system—without argument, the largest and most important aspect of GBA programming—so quickly is that you miss out on a lot of vital subjects such as interrupts, timers, and button input. These subjects are so vital that it is difficult to fully demonstrate the graphics system without them, and yet that would add a huge amount of complexity to the graphics code. Despite the discrepancy and apparent switch, these really are advanced subjects that would have been too difficult to explain in the first half of the book—so here we are!

This chapter will explain interrupts and how they work, how you can use them, and even how you can write them yourself. Then I talk about the all-important subject of timers, how to slow down your program to a consistent frame rate. This has been something of a problem in prior chapters (aside from using the vblank), but now you will have the means to correct it.

Have you have ever wondered how professional GBA programmers seem to have such fine control over the hardware? How their games just seem to run perfectly, smoothly, with accurate timing, consistent frame rates? I sure have! It has everything to do with this chapter, because these professional games are using interrupts and timers to keep the "machine" running smoothly. This subject will really help you to refine your GBA coding skills and make your code run smoothly and reliably.

Here are the subjects you'll learn about in this chapter:

- Using interrupts
- Using timers

Using Interrupts

Do you ever feel as if you are interrupted far too often when trying to get work done? I am constantly interrupted while writing code, writing the text of this chapter, and so on. There is a parallel in computerdom, and it takes the shape of either a hardware interrupt or a software interrupt. A hardware interrupt is a physical event that pauses the CPU while some other process (such as a memory copy) is occurring. For instance, a DMA memory copy causes a brief interrupt to occur, halting the CPU until it is finished. On the other hand, there are software interrupts, which are virtual interruptions of the program, all occurring within the CPU, rather than outside of it. A software interrupt is common in a multitasking operating system like Windows 2000 or XP. Since the GBA is a console video game machine, as you might have expected, all interrupts occur in the hardware side. The good news is that you can trigger one of these interrupts using a CPU or BIOS instruction.

An interrupt basically works like this. First, disable all interrupts, because if an interrupt occurs while you are screwing with the interrupt registers, your GBA could melt. Okay, not really, but it would probably look like your GBA is possessed because weird things could happen. In the Windows world, we call that a GPF, a general protection fault, meaning that the core has been corrupted. I have always thought of an operating system's core as the central armory in a medieval castle—the building inside the castle walls, surrounded by a courtyard, where merchants and farmers sell their goods.

Where was I? Oh yes, interrupts. After you have disabled interrupts, then you configure the interrupt registers before enabling the interrupts again. Think of it as telling the cannoneers atop your castle walls, "Don't you dare fire it while I'm reloading!"

The chief interrupt officer of the GBA is REG_IME, which has an unofficial title of "interrupt master enable register." When you want to disable interrupts, you set REG_IME = 0x0. Likewise, to enable interrupts, you set REG_IME = 0x1. If you simply set this register, nothing will happen, because you haven't specified which interrupts should occur—the who, what, when, where, and how, so to speak. To enable specific interrupts, you set specific bits in the interrupt enable register, REG_IE. This subordinate "enable interrupts" register works on each type of interrupt individually. There are interrupts available for DMA, vertical blank, horizontal blank, vertical count, timers, serial communications, and buttons, and each type of interrupt has a special bit value and a specific register. For instance, the interrupt register for DMA2 is REG_DMA2CNT, and the interrupt register for the display status is REG_DISPSTAT. Let's take a look at that one right now (see Table 8.1).

As I mentioned, REG_DISPSTAT is just the interrupt register for the display status, which is the most oft-used interrupt. Now, in order to actually turn on an interrupt, you need to know what REG_IE bits represent. Table 8.2 lists the bits for that register.

Table 8.1 REG_DISPSTAT Bits

Bit	Description
0	VB - vertical blank is occurring
1	HB - horizontal blank is occurring
2	VC - vertical count reached
3	VBE - enables vblank interrupt
4	HBE - enables hblank interrupt
5	VCE - enables vcount interrupt
6–15	VCOUNT - vertical count value (0–159)

Table 8.2 REG_IE Bits

Bit	Description
0	VB - vertical blank interrupt
1	HB - horizontal blank interrupt
2	VC - vertical scanline count interrupt
3	T0 - timer 0 interrupt
4	T1 - timer 1 interrupt
5	T2 - timer 2 interrupt
6	T3 - timer 3 interrupt
7	COM - serial communication interrupt
8	DMA0 - DMA0 finished interrupt
9	DMA1 - DMA1 finished interrupt
10	DMA2 - DMA2 finished interrupt
11	DMA3 - DMA3 finished interrupt
12	BUTTON - button interrupt
13	CART - game cartridge interrupt
14–15	unknown/unused

The REG_IE bits are used quite often and are needed in order to create any interrupt, so it is helpful to create some definitions of these bit values, as follows. The hexadecimal values in this list of definitions allow you to perform a bitwise AND with the REG_IE register in order to set the specific bit, without interfering with any of the other bits.

```
#define INT_VBLANK 0x0001
#define INT_HBLANK 0x0002
#define INT_VCOUNT 0x0004
#define INT_TIMER0 0x0008
#define INT_TIMER1 0x0010
#define INT_TIMER2 0x0020
#define INT_TIMER3 0x0040
#define INT_COM    0x0080
#define INT_DMA0   0x0100
#define INT_DMA1   0x0200
#define INT_DMA2   0x0400
#define INT_DMA3   0x0800
#define INT_BUTTON 0x1000
#define INT_CART   0x2000
```

It is pretty interesting how an interrupt actually occurs. What happens is that when an interrupt is triggered, the CPU saves the state of all the registers and then passes control to the interrupt service routine (which you must specify). After the ISR is finished, the CPU restores the registers and continues from the point where it was interrupted.

As for the ISR, that is something you must write yourself! Thankfully, it can be a C function, rather than assembler. The key to writing an ISR is understanding one simple fact: Every interrupt, regardless of type, is set to branch out to memory address 0x3007FFC. What you must do is intercept that memory address and have it point to your own ISR (a simple C function that I'll show you how to write). What I mean by "every interrupt" is exactly that, taken literally. All interrupts are passed to that memory address, so when an interrupt comes in, you must check to see which interrupt was triggered. The nice thing about this is that you need only write a single ISR for all the interrupts you are using in your GBA program.

There is another helper register called REG_IF that is a duplicate of REG_IE and is used to determine which interrupt was triggered (refer to Table 8.2 for the bit layout of REG_IF). However, don't be confused by this fact. One and *only* one interrupt will occur at a time! So the REG_IF register will have only one bit set, not several. You don't process all the interrupts that are occurring—that was a funny mistake I made when first learning about interrupts. It makes perfect sense if you think about it. Since the CPU is saving everything and sending control off to 0x3007FFC, why would there be more than one interrupt happening?

The good news about that is that you can simply compare REG_IF with the various interrupt definitions to see which one is occurring. More than likely, you will be using just one or two interrupts in your own programs, so a comprehensive check for all the interrupts is not necessary. Just look for the interrupt you have turned on, and that is all. For example:

```
if ((REG_IF & INT_TIMER0) == INT_TIMER0)
{
    //your timer code goes here
}
```

At the end of your ISR, be sure to turn off the bit for that particular interrupt request:

```
REG_IF |= INT_TIMER0
```

Let's take this pseudocode a step further and make it a little more complete before actually writing a complete program. First, let's define a register to the memory address for interrupts:

```
#define REG_INTERRUPT *(unsigned int*)0x3007FFC
```

So now, all you have to do is pass the name of your ISR function to this register, and the compiler will copy the address of that function into the interrupt link. Here's a short snippet that includes all the steps:

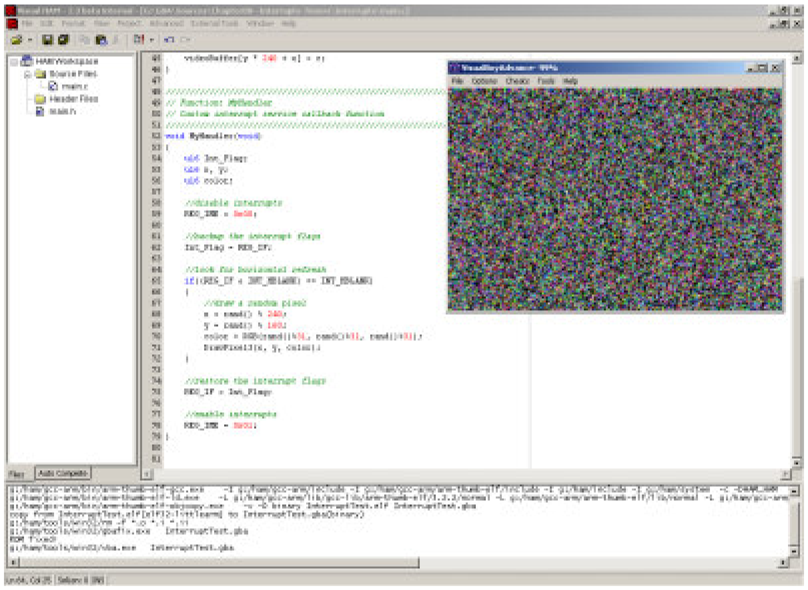
```
//first, turn off interrupts
REG_IME = 0x00;
//make ISR point to my own function
REG_INTERRUPT = (unsigned int)MyHandler;
//turn on vblank interrupt
REG_IE |= INT_VBLANK;
//tell dispstat about vblank interrupt
REG_DISPSTAT |= 0x08;
//lastly, turn interrupts back on
REG_IME = BIT00;
```

Now all that is needed is your own function for dealing with interrupts. Since I called it MyHandler in the preceding code, that's what I'll call it here. I have not commented this function, so as to keep it short. The full-blown handler in the InterruptTest program (a little further on) fully explains each line.

```
void MyHandler(void)
{
    REG_IME = 0x00;
    Int_Flag = REG_IF;
    if((REG_IF & INT_HBLANK) == INT_HBLANK)
    {
        //horizontal refresh--do something quick!
    }
    REG_IF = Int_Flag;
    REG_IME = 0x01;
}
```

The InterruptTest Program

The InterruptTest program (shown in Figure 8.1) demonstrates how to create an interrupt service routine in the form of a callback function. It is surprisingly easy to set up custom interrupts on the GBA, so the program is fairly short. This program does something very simple, because I want you to focus more on the interrupt code than any fancy display code or demo. Therefore, this program simply draws a mode 3 pixel when an interrupt occurs. I should point out that horizontal blank is a very short time interval that you shouldn't screw around with, or the display could go fubar (for the scientist, that means the horizontal blank has been distended, resulting in possible loss of image). Drawing a pixel is a one-liner, but drawing a random pixel is a three-liner, so it provides just enough to prove the interrupt is working.



```
    videoBuffer[y * 240 + x] = c;
}

//////////////////////////////////////////////////////////////
// Function: MyHandler
// Custom interrupt service callback function
//////////////////////////////////////////////////////////////
void MyHandler(void)
{
    u16 Int_Flag;
    u16 x, y;
    u16 color;

    //disable interrupts
    REG_IME = 0x00;

    //backup the interrupt flags
    Int_Flag = REG_IF;

    //look for horizontal refresh
    if((REG_IF & INT_HBLANK) == INT_HBLANK)
    {
        //draw a random pixel
        x = rand() % 240;
        y = rand() % 160;
        color = RGB(rand()%31, rand()%31, rand()%31);
        DrawPixel3(x, y, color);
    }

    //restore the interrupt flags
    REG_IF = Int_Flag;

    //enable interrupts
    REG_IME = 0x01;
}
```

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

Figure 8.1

The InterruptTest program demonstrates how to create a custom interrupt service routine.

The benefit here also is that you gain some experience working with the hblank, which is different from vblank, because there are 160 hblank interrupts for every one vblank, so your hblank code must be fast! In this example, what is happening is that 160 pixels are being sent to video memory every time the screen is refreshed. Some pixels are added behind the scanline and don't appear until the next vblank, while some pixels are added before the scanline and do appear right away. It's an interesting thing to play around with. I would recommend against using hblank unless absolutely necessary because it affects the performance of the video system.

The InterruptTest Header File

```
/////////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 8: Interrupts, Timers, and DMA
// InterruptTest Project
// main.h header file
/////////////////////////////////////////////////////////////

#ifndef _MAIN_H
#define _MAIN_H

#include <stdlib.h>

//define some data type shortcuts
typedef unsigned char u8;
typedef unsigned short u16;
typedef unsigned long u32;
typedef signed char s8;
typedef signed short s16;
typedef signed long s32;

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

//define some display registers
#define REG_DISPCNT *(u32*)0x4000000
```

```
#define BG2_ENABLE 0x400
#define SetMode(mode) REG_DISPCNT = (mode)

//define some interrupt registers
#define REG_IME        *(u16*)0x4000208
#define REG_IE         *(u16*)0x4000200
#define REG_IF         *(u16*)0x4000202
#define REG_INTERRUPT  *(u32*)0x3007FFC
#define REG_DISPSTAT   *(u16*)0x4000004

//create prototype for custom interrupt handler
void MyHandler(void);

//define some interrupt constants
#define INT_VBLANK 0x0001
#define INT_HBLANK 0x0002
#define INT_VCOUNT 0x0004
#define INT_TIMER0 0x0008
#define INT_TIMER1 0x0010
#define INT_TIMER2 0x0020
#define INT_TIMER3 0x0040
#define INT_COM 0x0080
#define INT_DMA0 0x0100
#define INT_DMA1 0x0200
#define INT_DMA2 0x0400
#define INT_DMA3 0x0800
#define INT_BUTTON 0x1000
#define INT_CART 0x2000

//create pointer to video memory
unsigned short* videoBuffer = (unsigned short*)0x6000000;

#endif
```

The InterruptTest Source File

Now for the main source code file of the InterruptTest program. This is a short program listing, thanks to the header file, allowing you to focus on exactly what is going on.

```
//////////////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 8: Interrupts, Timers, and DMA
// InterruptTest Project
// main.c source code file
//////////////////////////////////////////////////////////////////

#define MULTIBOOT int __gba_multiboot;
MULTIBOOT

#include "main.h"

//////////////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
//////////////////////////////////////////////////////////////////
int main(void)
{
    //Set mode 3 and enable the bitmap background
    SetMode(3 | BG2_ENABLE);

    //disable interrupts
    REG_IME = 0x00;

    //point interrupt handler to custom function
    REG_INTERRUPT = (u32)MyHandler;

    //enable hblank interrupt (bit 4)
    REG_IE |= INT_HBLANK;

    //enable hblank status (bit 4)
    REG_DISPSTAT |= 0x10;

    //enable interrupts
    REG_IME = 0x01;

    //endless loop
    while(1);
    return 0;
}

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

/////////////////////////////////////////////////////////////
// Function: MyHandler
// Custom interrupt service callback function
/////////////////////////////////////////////////////////////
void MyHandler(void)
{
    u16 Int_Flag;
    u16 x, y;
    u16 color;

    //disable interrupts
    REG_IME = 0x00;

    //backup the interrupt flags
    Int_Flag = REG_IF;
    //look for horizontal refresh
    if((REG_IF & INT_HBLANK) == INT_HBLANK)
    {
        //draw a random pixel
        x = rand() % 240;
        y = rand() % 160;
        color = RGB(rand()%31, rand()%31, rand()%31);
        DrawPixel3(x, y, color);
    }

    //restore the interrupt flags
    REG_IF = Int_Flag;

    //enable interrupts
    REG_IME = 0x01;
}
```

Using Timers

The subject of timers is perhaps the most important subject you can learn about the GBA, aside from graphics programming, because timers are critical to keeping the game running at a stable frame rate, and they come in handy when you want to insert a delay into the game (for instance, when scrolling text on the screen). The timing within the GBA is precise. For example, the refresh rate (the time it takes to draw all 160 scanlines) takes 280,896 CPU cycles (also called ticks). The vertical blank period is not the same as vertical refresh—the blank is the period of time during which the pixel "pointer" (for lack of a better term) is moved from the bottom-right back up to the top-left to start refreshing the screen again—using the video buffer. This vblank period is exactly 83,776 cycles. To be more precise still, the horizontal blank (hblank) takes 228 cycles, while a horizontal draw (hdraw) takes 1,004 cycles. These are incomprehensible time periods for the human mind to grasp—billionths of a second, or nanoseconds.

There are four timers built into the GBA, and they are each capable of handling 16-bit numbers, meaning the timers count from 0 to 65,535. The timers are based on the system clock. There are four frequencies available that you can set for the timers, as listed in Table 8.3.

Table 8.3 Timer Frequencies

Value	Frequency	Duration
0	16.78 MHz clock	Every 59.595 nanoseconds
1	64 cycles	Every 7.6281 microseconds
2	256 cycles	Every 15.256 microseconds
3	1,024 cycles	Every 61.025 microseconds

This table can be converted to a set of definitions to be used when setting up a timer:

```
#define TIMER_FREQUENCY_SYSTEM  0x0

#define TIMER_FREQUENCY_64      0x1

#define TIMER_FREQUENCY_256     0x2

#define TIMER_FREQUENCY_1024    0x3
```

There are a few things I need to go over about timers before you can create a timer using one of these four frequencies. I'm sure you're eager to get started, so I'll be brief with the following descriptions. First, you will need to select one or more timers to program. The four timers have the following definitions, which point to the time control memory addresses:

```
#define REG_TM0CNT *(volatile u16*)0x4000102

#define REG_TM1CNT *(volatile u16*)0x4000106

#define REG_TM2CNT *(volatile u16*)0x400010A

#define REG_TM3CNT *(volatile u16*)0x400010E
```

Now these are merely the addresses of where to change the status bits for the timer. To actually read the values generated by the timers, you'll need to look at a different set of memory addresses set aside for this purpose. These are called the REG_TMxD addresses and are defined here:

```
#define REG_TM0D       *(volatile u16*)0x4000100

#define REG_TM1D       *(volatile u16*)0x4000104

#define REG_TM2D       *(volatile u16*)0x4000108

#define REG_TM3D       *(volatile u16*)0x400010C
```

The structure of these 16-bit memory addresses are laid out a bit at a time in Table 8.4.

Table 8.4 REG_TMxCNT Bits

Bit	Description
0-1	Frequency
2	Overflow from previous timer
3-5	*Not used*
6	Overflow generates interrupt
7	Timer enable
8-15	*Not used*

The first two bits are set to one of the TIMER_FREQUENCY_x defines above, while the other three options are set with the following defines:

```
#define TIMER_OVERFLOW          0x4

#define TIMER_IRQ_ENABLE        0x40

#define TIMER_ENABLE            0x80
```

Timers are quite a bit easier to use than interrupts because there is no callback function to worry about (although I would point out that it is entirely possible to create an interrupt of a timer). The TimerTest program is a very good demonstration of using timers, including the use of the overflow (which means that when one timer reaches the 65,536 limit, it resets to 0 and increments the next timer, if so configured). Overflow is a very nice feature in the GBA, providing for some convenient timing mechanisms, although you may use variables just as well to keep track of this sort of thing, perhaps by performing a test such as this:

```
timer = REG_TM0D;

if (timer % 65536)

{

    //overflow--time to deal with it

}
```

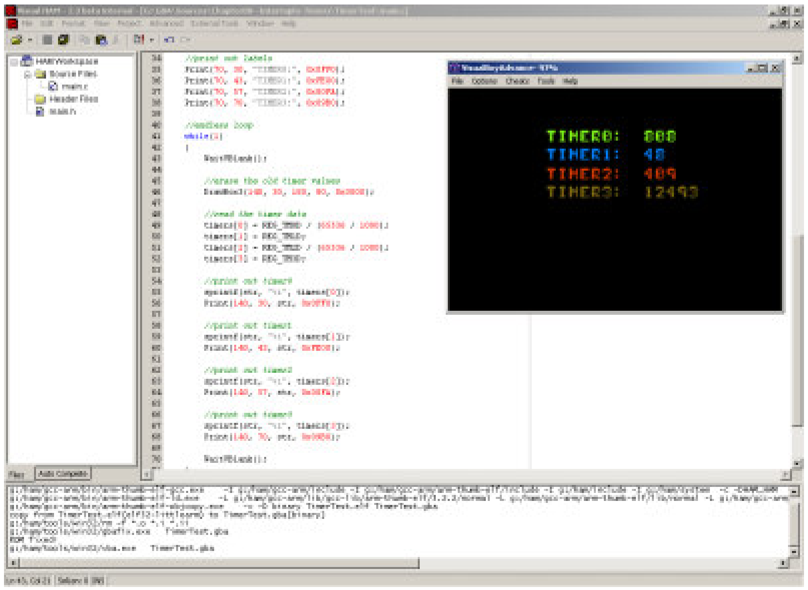
The TimerTest Program

And now to present the TimerTest program. Enjoy it while it lasts! Okay, the TimerTest program uses the font developed back in Chapter 5, "Bitmap-Based Video Modes," so you'll need the font.h file, which may be copied from the DrawText project folder from Chapter

5, or you may simply open the TimerTest project from \Sources\Chapter08\TimerTest. This program features both a header and source file, with the bulk of the GBA-specific hardware code hidden away in the header. Technically, this is a bad coding practice, but these programs are all so short, it would be silly to put just definitions, constants, and prototypes in a .h file, while moving all source listings into proper .c files. Therefore, the most commonly used functions are also included in the header.

That *is* the *proper* way to do it, after all, but it compiles all the same this way, so I prefer to keep things simple. Now, if you were to write you own GBA game, it would most likely be quite lengthy, so I would recommend using multiple source files for larger projects. You may even move graphics code into graphics.h and graphics.c, for instance, and then move the button code into button.h and button.c. It's entirely up to you—I present simple code listings, with each chapter and each project standing on its own so the reader may jump around at will, and leave organization up to you.

Now, about that TimerTest program. A screen shot is shown in Figure 8.2. Watch and learn.



```
TIMER0:  888
TIMER1:  48
TIMER2:  409
TIMER3:  12493
```

Figure 8.2

The TimerTest program displays four timers on the screen, two of which are overflows.

This project is like any other, with a main.c and main.h file. If you need a refresher on using Visual HAM, refer back to Chapter 3, "Game Boy Development Tools," for screen shots and a walk-through of creating projects and adding source files. The TimerTest header is listed first. Naturally, if you want to save some time, feel free to copy code from earlier projects and paste into each new project. I do reuse quite a bit of code from one project to the next.

The TimerTest Header

Here is the header for the TimerTest program, which should be saved in a file called main.h.

```
//////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 8: Using Interrupts and Timers
// TimerTest Project
// main.h header file
//////////////////////////////////////////////////

#ifndef _MAIN_H
#define _MAIN_H

#include <stdio.h>
#include <string.h>
#include "font.h"

//define some data type shortcuts
typedef unsigned char u8;
typedef unsigned short u16;
typedef unsigned long u32;
typedef signed char s8;
typedef signed short s16;
typedef signed long s32;

//declare some function prototypes
void DrawPixel3(int, int, unsigned short);
void DrawBox3(int, int, int, int, unsigned short)
void DrawChar(int, int, char, unsigned short);
void Print(int, int, char *, unsigned short);

//define the timer constants
#define TIMER_FREQUENCY_SYSTEM 0x0
#define TIMER_FREQUENCY_64 0x1
#define TIMER_FREQUENCY_256 0x2
#define TIMER_FREQUENCY_1024 0x3
#define TIMER_OVERFLOW 0x4
```

```
#define TIMER_ENABLE 0x80
#define TIMER_IRQ_ENABLE 0x40

//define the timer status addresses
#define REG_TM0CNT     *(volatile u16*)0x4000102
#define REG_TM1CNT     *(volatile u16*)0x4000106
#define REG_TM2CNT     *(volatile u16*)0x400010A
#define REG_TM3CNT     *(volatile u16*)0x400010E

//define the timer data addresses
#define REG_TM0D       *(volatile u16*)0x4000100
#define REG_TM1D       *(volatile u16*)0x4000104
#define REG_TM2D       *(volatile u16*)0x4000108
#define REG_TM3D       *(volatile u16*)0x400010C

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

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

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

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

//////////////////////////////////////////////////
// Function: DrawChar
// Draws a character one pixel at a time
//////////////////////////////////////////////////
void DrawChar(int left, int top, char letter, unsigned short color)
{
    int x, y;
    int draw;

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

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

```
}

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

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

//////////////////////////////////////////////////////////////
// Function: WaitVBlank
// Checks the scanline counter for the vertical blank period
//////////////////////////////////////////////////////////////
void WaitVBlank(void)
{
    while(!(*ScanlineCounter));
    while((*ScanlineCounter));
}

#endif
```

The TimerTest Source Code

The main.c source code file for the TimerTest program is next. This code should be straightforward enough to follow, since the bulk of the program is stored away in the header file.

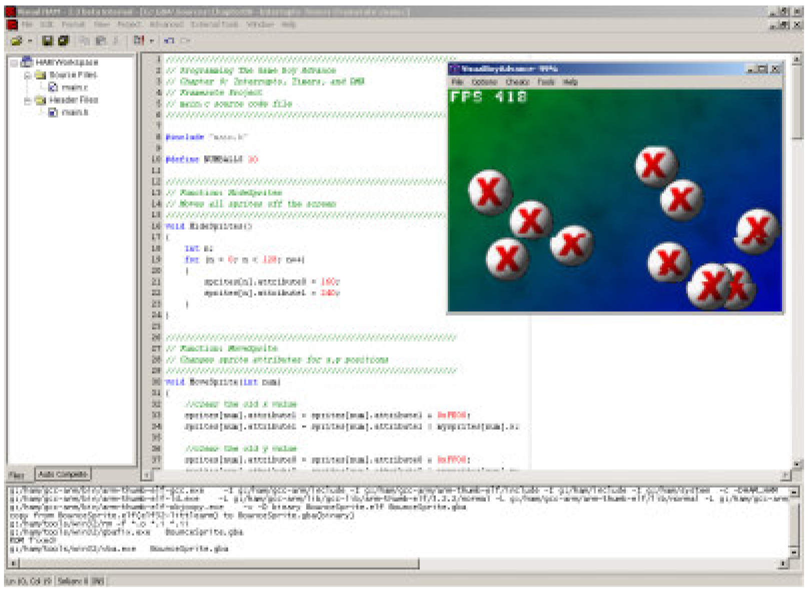
```
//////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 8: Using Interrupts and Timers
// TimerTest Project
// main.c source code file
//////////////////////////////////////////////////

#define MULTIBOOT int __gba_multiboot;
MULTIBOOT

#include "main.h"

//////////////////////////////////////////////////
// Function: main()
// Entry point for the program
//////////////////////////////////////////////////
int main()
{
    char str[20];
    int timers;

    //switch to video mode 3
    REG_DISPCNT = (3 | BG2_ENABLE);

    //turn on timer0, set to 256 clocks
    REG_TM0CNT = TIMER_FREQUENCY_256 | TIMER_ENABLE;

    //turn on timer1, grab overflow from timer0
    REG_TM1CNT = TIMER_OVERFLOW | TIMER_ENABLE;

    //turn on timer2, set to system clock
    REG_TM2CNT = TIMER_FREQUENCY_SYSTEM | TIMER_ENABLE;

    //turn on timer3, grab overflow from timer2
```

```
        REG_TM3CNT = TIMER_OVERFLOW | TIMER_ENABLE;

        //print out labels
        Print(70, 30, "TIMER0:", 0x0FF0);
        Print(70, 40, "TIMER1:", 0xFE00);
        Print(70, 60, "TIMER2:", 0x00FA);
        Print(70, 70, "TIMER3:", 0x09B0);


        //endless loop
        while(1)
        {
            WaitVBlank();

            //erase the old timer values
            DrawBox3(140, 30, 180, 80, 0x0000);

            //read the timer data
            timers[0] = REG_TM0D / (65536 / 1000);
            timers = REG_TM1D;
            timers = REG_TM2D / (65536 / 1000);
            timers = REG_TM3D;

            //print out timer0
            sprintf(str, "%i", timers[0]);
            Print(140, 30, str, 0x0FF0);

            //print out timer1
            sprintf(str, "%i", timers);
            Print(140, 40, str, 0xFE00);

            //print out timer2
            sprintf(str, "%i", timers);
            Print(140, 60, str, 0x00FA);
            //print out timer3
            sprintf(str, "%i", timers);
            Print(140, 70, str, 0x09B0);

            WaitVBlank();
        }

        return 0;
    }
```

The Framerate Program

The Framerate program was a long time waiting. I have wanted to delve into timers since the fourth chapter in order to display what the GBA is capable of doing in the graphics department. I think this program demonstrates that the VisualBoyAdvance emulator is working perfectly, for one thing, because a consistent frame rate of 60 FPS comes through when vblank is used. On the flip side, the frame rate skyrockets out of control with vblank turned off! This means one thing—you definitely want to try your code once in a while without vblank to see how it's doing without any chains attached!

Figure 8.3 shows the Framerate program running with vblank turned off. This is basically the AnimSprite program from the previous chapter, which was the perfect example of a situation where knowing the frame rate would be extremely useful, as this was the most graphically intense program of the book so far. Look at that frame rate!

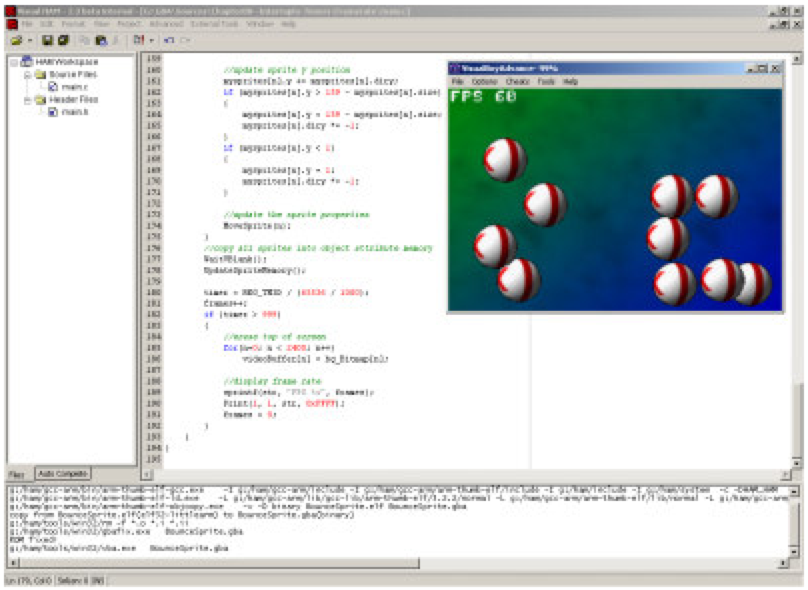


```
//////////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 8: Interrupts, Timers, and DMA
// Framerate Project
// main.c source code file
//////////////////////////////////////////////////////////////

#include "main.h"

#define NUMBALLS 10

//////////////////////////////////////////////////////////////
// Function: HideSprites
// Moves all sprites off the screen
//////////////////////////////////////////////////////////////
void HideSprites()
{
    int n;
    for (n = 0; n < 128; n++)
    {
        sprites[n].attribute0 = 160;
        sprites[n].attribute1 = 240;
    }
}

//////////////////////////////////////////////////////////////
// Function: MoveSprite
// Changes sprite attributes for x,y positions
//////////////////////////////////////////////////////////////
void MoveSprite(int num)
{
    //clear the old x value
    sprites[num].attribute1 = sprites[num].attribute1 & 0xFE00;
    sprites[num].attribute1 = sprites[num].attribute1 | mysprites[num].x;

    //clear the old y value
    sprites[num].attribute0 = sprites[num].attribute0 & 0xFF00;
```

Figure 8.3

The Framerate program running with vblank turned off results in very high frame rates in both the emulator and an actual GBA. Note the image tearing, though!

Turning vblank back on (by uncommenting the WaitVBlank() line) results in a consistent 60 FPS (as shown in Figure 8.4), which is what you might expect under a video system that has a vertical refresh of 60 Hz. Now there are some weird things you can do to get around the 60 FPS limit without experiencing image tearing (as evidenced in Figure 8.3).



```
        //update sprite y position
        mysprites[n].y += mysprites[n].diry;
        if (mysprites[n].y > 159 - mysprites[n].size)
        {
            mysprites[n].y = 159 - mysprites[n].size;
            mysprites[n].diry *= -1;
        }
        if (mysprites[n].y < 1)
        {
            mysprites[n].y = 1;
            mysprites[n].diry *= -1;
        }

        //update the sprite properties
        MoveSprite(n);
    }
    //copy all sprites into object attribute memory
    WaitVBlank();
    UpdateSpriteMemory();

    timer = REG_TM3D / (65536 / 1000);
    frames++;
    if (timer > 999)
    {
        //erase top of screen
        for(n=0; n < 2400; n++)
            videoBuffer[n] = bg_Bitmap[n];

        //display frame rate
        sprintf(str, "FPS %i", frames);
        Print(1, 1, str, 0xFFFF);
        frames = 0;
    }
  }
}
```

Figure 8.4

With vblank in use, the Framerate program reports a nice consistent 60 FPS, as expected, with nicely-drawn sprites.

This program is familiar if you have already worked through Chapter 7, "Rounding Up Sprites." If you are jumping around from chapter to chapter out of order, you'll have no problem running the program in this incarnation, because it is listed in its entirety. It might have been possible to just point out the differences between this Framerate program and the AnimSprite program from the last chapter, but there were significant changes to both the header and source code file, so I decided to just list both here in their entirety.

If you wish, you may load this project directly off the CD-ROM, which may be a good idea if you already worked through the AnimSprite project. It is located in \Sources\Chapter08\Framerate.

The Framerate Header

Now here is the header file, main.h, which is included by the main.c file. This file basically hides away all the messy details of a GBA program, allowing the main source code file, main.c, to stick to the goal and is also less distracting.

```
//////////////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 8: Using Interrupts and Timers
// Framerate Project
// main.h header file
//////////////////////////////////////////////////////////////////
```

```
#ifndef _MAIN_H
#define _MAIN_H

//define some data type shortcuts
typedef unsigned char u8;
typedef unsigned short u16;
typedef unsigned long u32;
typedef signed char s8;
typedef signed short s16;
typedef signed long s32;

#include <stdlib.h>
#include <stdio.h>
#include "bg.raw.c"
#include "ball.h"
#include "font.h"

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

//define the timer constants
#define TIMER_FREQUENCY_SYSTEM 0x0
#define TIMER_FREQUENCY_64 0x1
#define TIMER_FREQUENCY_256 0x2
#define TIMER_FREQUENCY_1024 0x3
#define TIMER_OVERFLOW 0x4
#define TIMER_ENABLE 0x80
#define TIMER_IRQ_ENABLE 0x40

//define the timer status addresses
```

```
#define REG_TM0CNT      *(volatile u16*)0x4000102

#define REG_TM1CNT      *(volatile u16*)0x4000106

#define REG_TM2CNT      *(volatile u16*)0x400010A

#define REG_TM3CNT      *(volatile u16*)0x400010E


//define the timer data addresses

#define REG_TM0D        *(volatile u16*)0x4000100

#define REG_TM1D        *(volatile u16*)0x4000104

#define REG_TM2D        *(volatile u16*)0x4000108

#define REG_TM3D        *(volatile u16*)0x400010C


//macro to change the video mode

#define SetMode(mode) REG_DISPCNT = (mode)


//create a pointer to the video buffer

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


//define some video addresses

#define REG_DISPCNT *(volatile unsigned short*)0x4000000

#define BGPaletteMem ((unsigned short*)0x5000000)



//declare scanline counter for vertical blank

volatile u16* ScanlineCounter = (volatile u16*)0x4000006;


//define object attribute memory state address

#define SpriteMem ((unsigned short*)0x7000000)


//define object attribute memory image address

#define SpriteData ((unsigned short*)0x6010000)


//video modes 3-5, OAMData starts at 0x6010000 + 8192

unsigned short* SpriteData3 = SpriteData + 8192;
```

```
//define object attribute memory palette address
#define SpritePal ((unsigned short*)0x5000200)

//misc sprite constants
#define OBJ_MAP_2D 0x0
#define OBJ_MAP_1D 0x40
#define OBJ_ENABLE 0x1000
#define BG2_ENABLE0x400

//attribute0 stuff
#define ROTATION_FLAG    0x100
#define SIZE_DOUBLE       0x200
#define MODE_NORMAL       0x0
#define MODE_TRANSPARENT  0x400
#define MODE_WINDOWED     0x800
#define MOSAIC            0x1000
#define COLOR_256         0x2000
#define SQUARE            0x0
#define TALL              0x4000
#define WIDE              0x8000

//attribute1 stuff
#define SIZE_8            0x0
#define SIZE_16           0x4000
#define SIZE_32           0x8000
#define SIZE_64           0xC000

//an entry for object attribute memory (OAM)
typedef struct tagSprite
{
    unsigned short attribute0;
    unsigned short attribute1;
    unsigned short attribute2;
    unsigned short attribute3;
}Sprite,*pSprite;

//create an array of 128 sprites equal to OAM
Sprite sprites[128];

typedef struct tagSpriteHandler
{
    int alive;
    int x, y;
    int dirx, diry;
    int size;

}SpriteHandler;

SpriteHandler mysprites[128];

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

//////////////////////////////////////////////////
// Function: DrawChar
// Draws a character one pixel at a time
//////////////////////////////////////////////////
```

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

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

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

/////////////////////////////////////////////////////////////
// Function: WaitVBlank
// Checks the scanline counter for the vertical blank period
/////////////////////////////////////////////////////////////
void WaitVBlank(void)
{
    while(!(*ScanlineCounter));
    while((*ScanlineCounter));
}
```

```
#endif
```

The Framerate Source

Now for the main source code file for the Framerate program. You will likely see a lot of familiar code here, but there is also a lot of new code due to the use of timers to determine the frame rate. Most of the important code is located at the end of the main game loop, just after the WaitVBlank function call.

```
/////////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 8: Using Interrupts and Timers
// Framerate Project
// main.c source code file
/////////////////////////////////////////////////////////////

#define MULTIBOOT int __gba_multiboot;
MULTIBOOT

#include "main.h"

#define NUMBALLS 10

/////////////////////////////////////////////////////////////
// Function: HideSprites
// Moves all sprites off the screen
/////////////////////////////////////////////////////////////
void HideSprites()
{
    int n;
    for (n = 0; n < 128; n++)
    {
        sprites[n].attribute0 = 160;
        sprites[n].attribute1 = 240;
    }
}

////////////////////////////////////////////////////////////
// Function: MoveSprite
// Changes sprite attributes for x,y positions
////////////////////////////////////////////////////////////
void MoveSprite(int num)
{
    //clear the old x value
    sprites[num].attribute1 = sprites[num].attribute1 & 0xFE00;
    sprites[num].attribute1 = sprites[num].attribute1 | mysprites[num].x;

    //clear the old y value
    sprites[num].attribute0 = sprites[num].attribute0 & 0xFF00;
    sprites[num].attribute0 = sprites[num].attribute0 | mysprites[num].y;
}

////////////////////////////////////////////////////////////
// Function: UpdateSpriteMemory
// Copies the sprite array into OAM memory
////////////////////////////////////////////////////////////
void UpdateSpriteMemory(void)
{
    int n;
    unsigned short* temp;
    temp = (unsigned short*)sprites;
    for(n = 0; n < 128 * 4; n++)
        SpriteMem[n] = temp[n];
}

/////////////////////////////////////////////////////////////
// Function: InitSprite
// Initializes a sprite within the sprite handler array
/////////////////////////////////////////////////////////////
```

```
void InitSprite(int num, int x, int y, int size, int color, int tileIndex)
{
    unsigned int sprite_size = 0;

    mysprites[num].alive = 1;
    mysprites[num].size = size;
    mysprites[num].x = x;
    mysprites[num].y = y;

    //in modes 3-5, tiles start at 512, modes 0-2 start at 0
    sprites[num].attribute2 = tileIndex;

    //initialize
    sprites[num].attribute0 = color | y;

    switch (size)
    {
        case 8: sprite_size = SIZE_8; break;
        case 16: sprite_size = SIZE_16; break;
        case 32: sprite_size = SIZE_32; break;
        case 64: sprite_size = SIZE_64; break;
    }

     sprites[num].attribute1 = sprite_size | x;
}

/////////////////////////////////////////////////////////////
// Function: UpdateBall
// Copies current ball sprite frame into OAM
/////////////////////////////////////////////////////////////
void UpdateBall(index)
{
    u16 n;
    //copy sprite frame into OAM
    for(n = 0; n < 512; n++)
        SpriteData3[n] = ballData[(512*index)+n];
}

/////////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
/////////////////////////////////////////////////////////////
int main()
{
    char str[10];
    int n;
    int frames = 0;
    int timer = 0;

    //set the video mode--mode 3, bg 2, with sprite support
    SetMode(3 | OBJ_ENABLE | OBJ_MAP_1D | BG2_ENABLE);

    //draw the background
    for(n=0; n < 38400; n++)
    videoBuffer[n] = bg_Bitmap[n];

    //set the sprite palette
     for(n = 0; n < 256; n++)
         SpritePal[n] = ballPalette[n];

    //move all sprites off the screen
    HideSprites();

    //initialize the balls--note all sprites use the same image (512)
    for (n = 0; n < NUMBALLS; n++)
    {
        InitSprite(n, rand() % 230, rand() % 150, ball_WIDTH,
            COLOR_256, 512);
        while (mysprites[n].dirx == 0)
            mysprites[n].dirx = rand() % 6 - 3;
        while (mysprites[n].diry == 0)
            mysprites[n].diry = rand() % 6 - 3;
    }

    int ball_index=0;

    //start the timer
    REG_TM3CNT = TIMER_FREQUENCY_256 | TIMER_ENABLE;



    //main loop
    while(1)
    {
        //increment the ball animation frame
        if(++ball_index > 31)ball_index=0;
        UpdateBall(ball_index);

        for (n = 0; n < NUMBALLS; n++)
        {
            //update sprite x position
            mysprites[n].x += mysprites[n].dirx;
            if (mysprites[n].x > 239 - mysprites[n].size)
            {
                mysprites[n].x = 239 - mysprites[n].size;
                mysprites[n].dirx *= -1;
            }
            if (mysprites[n].x < 1)
            {
                mysprites[n].x = 1;
                mysprites[n].dirx *= -1;
            }
            //update sprite y position
            mysprites[n].y += mysprites[n].diry;
            if (mysprites[n].y > 159 - mysprites[n].size)
            {
                mysprites[n].y = 159 - mysprites[n].size;
                mysprites[n].diry *= -1;
            }
            if (mysprites[n].y < 1)
            {
                mysprites[n].y = 1;
                mysprites[n].diry *= -1;
            }

            //update the sprite properties
            MoveSprite(n);
        }
        //copy all sprites into object attribute memory
        WaitVBlank();
        UpdateSpriteMemory();

        timer = REG_TM3D / (65536 / 1000);
        frames++;
        if (timer > 999)
        {
            //erase top of screen
            for(n=0; n < 2400; n++)
              videoBuffer[n] = bg_Bitmap[n];

            //display frame rate
            sprintf(str, "FPS %i", frames);
            Print(1, 1, str, 0xFFFF);
            frames = 0;
        }
```

```
        }
}
```

It's nice to know how the program is performing by displaying the frame rate, so I'm sure you'll find a use for this code in many a game. It could be optimized quite a bit, I won't deny it—particularly the code that erases the top of the screen. However, that doesn't seem to be affecting the frame rate at all. Really, from the screen shots, it is apparent that WaitVBlank is a serious detriment to the actual capabilities of the GBA! The CPU is capable of handling much more than these small example programs, which aren't even pushing the hardware. As soon as you start to see the frame rate drop below 60 FPS, then it's time to look into optimizations. But until then, don't waste time on it, and just keep working away on whatever game you are creating! After all, remember that optimization comes last, and only if it's needed. Worry more about writing good clean code first, and work on making an enjoyable game, because in all likelihood you won't tap the power of the GBA.

Summary

This chapter provided the theory and practical sides of using interrupts and timers to enhance your GBA programs. Using these two key hardware facilities, you will be able to better control the granularity of your programs—that is, how fast or slow they run, how smoothly the screen is refreshed, and how to process code based on certain interrupts. Not only did you learn how to use interrupts and timers, you have learned the practical use for them by writing an animated sprite program that displays the frame rate. As this is an extremely useful new feature, I'm sure you'll find a need for it in all of your own GBA projects.

Challenges

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

Challenge 1: The InterruptTest program draws a pixel every time during every hblank. Modify the program so it instead draws a box during every vertical blank instead.

Challenge 2: The TimerTest program displays the values of each of the four timers, two of which are set to specific frequencies, the other two set to overflow. Modify the program using different sets of frequencies and note the change in the timers displayed on the screen.

Challenge 3: The Framerate program displays a consistent 60 FPS. Bump up the number of sprites displayed by modifying the NUMBALLS constant, adding 10 balls to the number each time, and note the change in the frame rate as the ball count increases. Try to determine the maximum number of balls that can be animated before the frame rate drops below 60 FPS.

Chapter Quiz

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

1. What is the interrupt master enable register that turns interrupts on or off?
    - A. REG_IE
    - B. REG_IF
    - C. REG_IME
    - D. REG_DISPSTAT

2. How many interrupts are available on the GBA?
    - A. 14
    - B. 4
    - C. 8
    - D. 12

3. What register is used to enable interrupt status for vblank, hblank, and vcount?
    - A. REG_IME
    - B. REG_DISPSTAT
    - C. REG_INTERRUPT
    - D. REG_CODE

4. To what memory address does the CPU shift control during an interrupt?
    - A. 0x6000000
    - B. 0x7001000
    - C. 0x4F00401
    - D. 0x3007FFC

5. What register is used in a custom interrupt callback function to determine which interrupt has occurred?
    - A. REG_IF
    - B. REG_IE
    - C. REG_DISPCNT
    - D. REG_IME

6. Which interrupt, if enabled, is triggered 160 times for every screen refresh?
    - A. INT_DMA1
    - B. INT_HBLANK

C. INT_TIMER3
D. INT_VBLANK

7. True/False: Does the LCD screen on the GBA handle screen refresh itself?
    A. True
    B. False

8. How many CPU cycles are used up for every vertical refresh of the screen?
    A. 280,896
    B. 4,836,938,238
    C. 160
    D. 16,386

9. How many timers are available in the GBA?
    A. 1
    B. 2
    C. 3
    D. 4

10. What is the largest numeric value that a 16-bit timer can handle?
    A. 4,096
    B. 16,384
    C. 65,536
    D. 1,048,576