




# Chapter 9

# The Sound System



**T**his chapter covers the fascinating subject of sound playback on the Game Boy Advance, with coverage of the sound hardware, digital sound formats, and the code to play sound samples. There are two sample programs in this chapter that show how to use the sound hardware on the GBA, from a simple playback program to a more elaborate program that uses button input to play several sounds. Are you ready to jump into the code and get started? This is a pretty fast-paced chapter that gets down to the metal and shows you exactly what you need to play sound. You will be adding sound to your own games in no time.

Here are the main topics of this chapter:

- Introduction to sound programming
- Playing digital sound files
- The PlaySamples program

# Introduction to Sound Programming

The GBA has a very good sound system that was well designed and is perfectly suitable for handheld games. While most gamers just use the built-in speaker, the GBA does support stereo sound when using headphones. This is due to the dual digital channels in the sound chip. Ideally, you will want to wear headphones while playing games in order to take advantage of stereo sound, because the built-in speaker simply combines the two channels, dropping the stereo effect. If you have not tried it with headphones, you'll be surprised by how much better the games are in stereo.

## GBA Sound Hardware

The GBA has two 8-bit digital-to-analog converters (DACs) for playing digital sound effects and music. These two channels, which are referred to as direct sound, support 8-bit signed samples. In addition, the GBA is backward compatible with previous Game Boy models, so it includes the earlier four sound channels. The two channels are called Direct Sound A and Direct Sound B and are capable of playing back 8-bit signed PCM samples. Pulse code modulation (PCM) is a raw format that is supported by most sound editor programs and may be saved as a .wav file.

## FM Synthesis Support

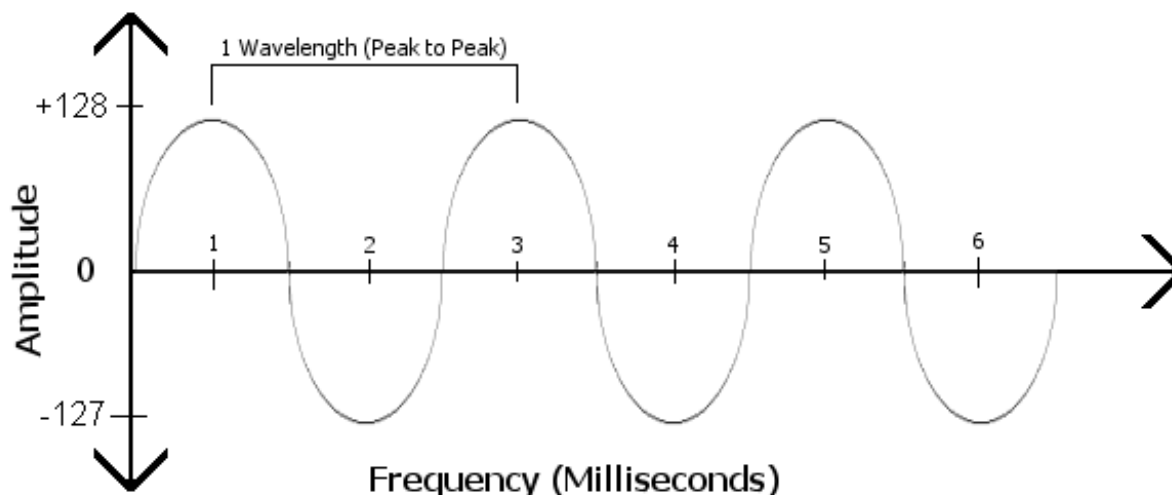
The sound chip is backward compatible with Game Boy Color, providing four FM sound channels. Frequency modulation (FM) synthesis is a method of alternating the frequency of a sine wave at fast intervals to produce sound effects and music. The result is not bad, but how can I explain the output? It sounds fuzzy, like there is white noise in the sound, as in an improperly tuned radio station or TV channel—quite different from digital sound. Since I can't imagine any practical use for the four FM channels in the GBA, throwbacks to a previous decade, I am going to focus exclusively on the two direct sound channels. I hope you understand my reasoning, because there is no need for FM synthesis when you have a DAC available! That is akin to preferring an Apple II over a Pentium 4 PC—without considering the novelty, that is. For all practical purposes, just ignore the compatibility sound channels on the GBA, and focus on digital sound. There is no comparison.

However, I don't want to dismiss FM sound completely, because there are cases where it can be helpful, in some circumstances where digital sound is overkill. For instance, FM is great for doing some kinds of sound effects, such as an airplane engine (which runs continuously), or for the sound of wind perhaps.

## Using Direct Sound for Digital Playback

Frequency modulation does work well to simulate sound mixing and does sound pretty good, in the context of small handheld games. In comparison, though, FM is simulated sound, rather than real sound. The reason for the poor sound quality in earlier Game Boy models was that they did not have the luxury of a DAC (whereas the GBA has two of them). With the GBA you can create sound effects yourself using a wave editor tool like Syntrillium's Cool Edit 2000 (included on the CD-ROM under \Tools\Cool Edit 2000), or you can download some

public domain wave files off the Web to use in your games. There is a tool for converting a wave file to a C source file and then storing the wave file's bytes inside a C array, in the same manner that bitmaps are converted. I will show you how to do it a little later in this chapter. Take a look at Figure 9.1 for an illustration of a waveform.



*Figure 9.1 - The sound produced by a waveform is determined by frequency and amplitude.*

## Sound Mixing

There is one issue with the GBA's sound capabilities, and it's not a problem at all once you understand the sound hardware. The GBA is capable of only a single digital sound at a time (per channel), with no support for mixing. What at first seems to be a problem, however, is really only the norm. Your PC doesn't have a hardware sound mixer either! Of course, a PC sound card can output CD music along with digital sound produced by a program (such as Windows Media Player or WinAmp), that level of hardware mixing does not translate to games at all. Indeed, the latest fast-paced game for the PC must do sound mixing on its own, as that is not built into the PC. Now, before you object, what I mean is that the *game engine* (such as direct sound) does the sound mixing, which for all practical purposes is a function of the game, while DirectX just happens to be installed separately.

The GBA has a very good sound chip built in that is at least on par with the early PC sound cards, which is fantastic for a handheld—while lacking such obvious things as Dolby™ ProLogic™, Dolby DTS™, Dolby Surround Sound™. I hope that earned a chuckle, because obviously such support is useless coming through the built-in speaker or headphones. The dual digital channels on the GBA are perfect for the types of games developed for it. Fortunately, HAM comes with an excellent sound-mixing library called Krawall.

Krawall is a complete sound engine for the GBA, providing everything you will need for a complete game sound solution, with an emphasis on speed, high-quality playback, and a



straightforward API. Although HAM includes the free version of Krawall, I encourage you to peruse the Krawall Web site at <http://mind.riot.org/krawall> and download the latest version with documentation and learn how to use it.

Krawall is free for personal use but does require a license for commercial use. Also, the free version is not as powerful as the fully licensed version. If you are serious about GBA sound, then you need to get at least a personal licensed copy for your own use and should purchase a commercial license without a second thought if you are an officially licensed GBA developer. In addition to mixing wave samples, Krawall also features a ProTracker module player that can play .mod, .xm, and .s3m music files flawlessly in the background while playing sound effects in the "foreground."

Building a sound mixer is somewhat beyond the goals of this single chapter, so I encourage you to look into Krawall as a solution. There are other sound libraries available for the GBA, which I have listed in Appendix B, "Recommended Books and Web Sites."

## Playing Digital Sound Files

Digital playback on the GBA is somewhat involved to the uninitiated, but the actual source code is not difficult to write and is definitely manageable in 20-30 lines of code. I will go over the specifics of the sound system and describe the registers and defines you will need to write a sound playback function. For starters, I'll walk you through a simple SoundTest program, which plays a single sample and ends. After you have gained an understanding of simple playback, I'll show you a program called PlaySamples, which plays several sounds based on button input.

## Playing Digital Sounds

The key to sound playback on the GBA is not getting the sound going, but how to make it stop when the sample is finished. The GBA doesn't inherently know when it has reached the end of a sample. There are two methods of controlling playback: DMA and interrupts. DMA is the preferred method, because it doesn't require any intervention from the programmer. Once the sample is started, the DMA controller automatically feeds the sound buffer. Interrupt-driven sound, on the other hand, requires the programmer to feed the sound buffer at each interval (which is usually during the vblank).

Wave samples for a GBA game must be converted to a C array and should be loaded consecutively into the direct sound memory buffer called `REG_FIFO_A`, which starts at address `0x4000A0`. Granted, there are always other ways to solve a problem, and in the case of GBA sound, you could convert a wave file to a binary file and link it into the program, although it's not a great solution when you are just learning this material.

Sound playback is interesting on the GBA. The 8-bit signed sound samples (which have a value range of only -127 to 128) are played back by the GBA sound chip in a first-in first-out (FIFO) process, meaning that lower address bytes are played first. While it might sound at first that sound playback goes in reverse, that is only a matter of how you perceive memory. I perceive memory being laid out linearly, 1 byte at a time, down a very long line (per-

haps like a train).

Some imagine computer memory in a 2D or even 3D layout, but I submit that the linear analogy more closely resembles the actual state. It is true that a memory chip is filled with nanoscopic transistors, or gates, in a grid and is even layered, so the perception of a 2D or 3D memory chip is accurate from a hardware perspective. However, we aren't electrical engineers—or at least, I'm not!—so it is the software standpoint that matters here. The one-dimensional line analogy will help you to better understand how software works, if you have never really thought about it in detail. A wave file or bitmap file converted to a C array is a linear array of bytes, which might be thought of as pure bits. Now, a sound sample comprises just 1 signed byte, but it is the linear playback of many sample bytes, sent through the DAC, that produces a digital sound.

There are two possible ways to play a sample: using either DMA or an interrupt. DMA mode is more efficient because consecutive samples are automatically loaded without interruption to the game. The interrupt mode must briefly pause the program to load the FIFO buffer but is possibly easier to use, and perhaps even necessary to use, in some cases.

The direct sound channels are controlled by the `REG_SOUND_CNT_H` register located at memory address `0x04000082`, which has the layout shown in Table 9.1.

**Table 9.1** `REG_SOUND_CNT_H` Bits

Bits	Description
0-1	Channel 1-4 volume control
2	Direct Sound A volume control
3	Direct Sound B volume control
4-7	<i>Unused</i>
8	Enable Direct Sound A to right speaker
9	Enable Direct Sound A to left speaker
10	Direct Sound A sampling rate timer select
11	Direct Sound A reset FIFO
12	Enable Direct Sound B to right speaker
13	Enable Direct Sound B to left speaker
14	Direct Sound B sampling rate timer select
15	Direct Sound B reset FIFO

Since you will need a list of defines in order to program the sound channels, I'll provide that a little prematurely right now:

```

#define SND_ENABLED          0x00000080
#define SND_OUTPUT_RATIO_25 0x0000
#define SND_OUTPUT_RATIO_50 0x0001
#define SND_OUTPUT_RATIO_100 0x0002
#define DSA_OUTPUT_RATIO_50 0x0000
#define DSA_OUTPUT_RATIO_100 0x0004
#define DSA_OUTPUT_TO_RIGHT 0x0100
#define DSA_OUTPUT_TO_LEFT 0x0200
#define DSA_OUTPUT_TO_BOTH 0x0300
#define DSA_TIMER0          0x0000
#define DSA_TIMER1          0x0400
#define DSA_FIFO_RESET      0x0800
#define DSB_OUTPUT_RATIO_50 0x0000
#define DSB_OUTPUT_RATIO_100 0x0008
#define DSB_OUTPUT_TO_RIGHT 0x1000
#define DSB_OUTPUT_TO_LEFT 0x2000
#define DSB_OUTPUT_TO_BOTH 0x3000
#define DSB_TIMER0          0x0000
#define DSB_TIMER1          0x4000
#define DSB_FIFO_RESET      0x8000

```

The sound hardware is quite helpful when it comes to actually playing the sound sample. All you have to do (after copying the sample into the appropriate memory address for playback) is tell Direct Sound A or B to watch a specific timer for an overflow. You learned about timers in the previous chapter, which was kind of convenient, right? Well, it was planned that way. <Smile.> If you skipped over Chapter 8, "Using Interrupts and Timers," I recommend that you go back to it and first learn how timers and interrupts work before proceeding any further into this chapter.

When the specified timer (0 or 1) overflows, Direct Sound A or B will send another byte from the FIFO to the DAC for playback. The key is setting up the timers to the specific sampling rate of the wave file so it sounds right. Remember that the timers are used to play back the sound at the correct rate to accurately reproduce the sound. The way you can determine how to set the timers is by calculating how often a sample should be sent to the DAC, and this is based on the CPU cycles. At 16.7 MHz, the CPU has 16,777,216 cycles per second; this is a fixed value that you can count on in your GBA games, because the architecture of a console is fixed. To determine the number of cycles per sample, simply divide 16,777,216 by the sampling rate.

For example, suppose you want to play back a sample at 44.1 kHz, which is CD-quality music. Granted, this would take an enormous amount of memory, and you wouldn't want to do this in practice, so let's just call this a hypothetical situation.  $16,777,216 / 44,100 = 380.44$ , so you would want to set a timer to send a sample to the DAC every 380 CPU cycles. A sampling rate of one-fourth CD quality is more realistic for GBA sounds, so let's calculate

it.  $16,777,216 / 11,025 = 1,521.74$ , or rather, 1,521 cycles per sample. As you can see, the sound playback itself doesn't require much of the CPU's time, although the overhead of using timers and copying samples into memory does take a few more cycles.

Now, how would you go about setting up a timer to send a sample to the DAC every 1,521 CPU cycles? The timers are 16-bit, meaning they have a range of up to 65,535, with a selectable frequency of 1, 64, 256, or 1,024 CPU cycles. To program the counter, which will return to the preset value after an overflow, simply subtract the cycles from 65,535 to set the initial value of the timer. So, an 11 kHz sample would require a timer set to  $65,535 - 1,521 = 64,014$ .

## The SoundTest Program

Sound programming is not easy to explain as it is, and when dealing with registers and binary numbers, it can be quite confusing. To put this all into perspective, I've written a program called SoundTest, which simply plays a sound sample called *splash*, which is converted from splash.wav to splash.c and included in the main.c file. The program is only about a page in length, and the actual sound code is 20 or so lines. Figure 9.2 shows the program, although there is nothing displayed on the screen, as this program simply outputs the sound and then ends.

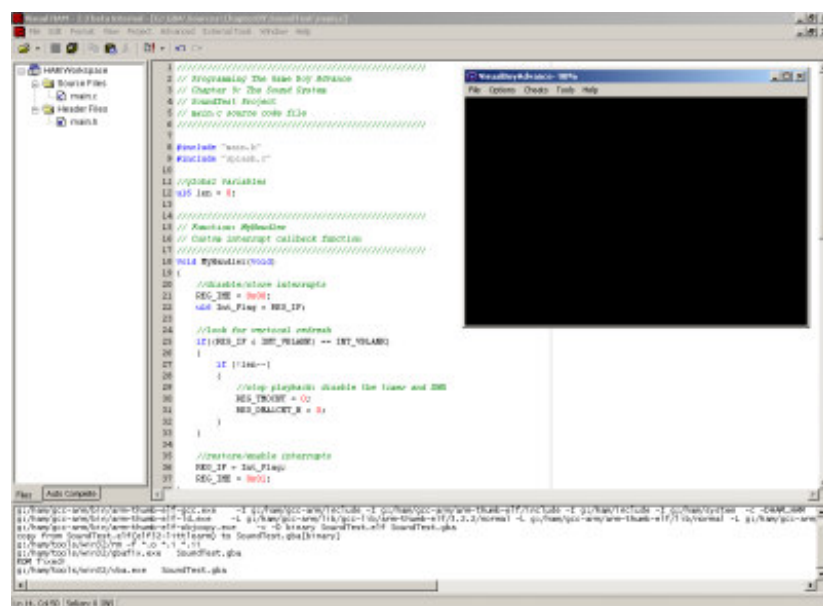


Figure 9.2  
The SoundTest program plays a digital sound sample.

## Converting the Sound File

In order to play a digital sample, a wave file must be converted into the raw binary format that the GBA can recognize and use. As explained earlier, that format is PCM and is stored in a .wav file. You are free to use any sound editing or converting program you like, but I prefer Cool Edit 2000. Some of the .wav files that I have are not all in PCM format. Here are some of the wave formats (or rather, audio codecs) that you are likely to encounter:



- A/mu-Law Wave
- ACM Waveform
- DVI/IMA ADPCM
- Microsoft ADPCM
- Windows PCM

As long as your sound-editing software is able to save files in the Windows PCM format, then the GBA will be able to play it. If it's not in the correct format, the converter program will print out an error message and fail to convert the file.

The program I have used to convert a wave file for use on the GBA is `wav2gba`, written by Rafael Vuijk (a.k.a. Dark Fader), and may be downloaded from <http://darkfader.net/gba>. I have included the `wav2gba.exe` program in each of the project folders for this chapter under `\Sources\Chapter09` on the CD-ROM, as well as in the `\Tools` folder. The `wav2gba` program is a command-line tool, just like the `gfx2gba` program you have used to convert graphics in previous chapters. `Wav2gba` has this syntax:

```
wav2gba <input.wav> <output.bin>
```

If you open a Command Prompt window (Start, Programs, Accessories menu), change to the folder for the `SoundTest` program (using the `CD` command), which is located in `\Sources\Chapter09\SoundTest` on the CD-ROM. You will of course want to copy the sources off the CD-ROM to your hard drive and then remove the read-only property from `\Sources` and all subfolders and files (simply right-click on `\Sources` and select Properties, then uncheck the Read Only check box).

Assuming you are in the `SoundTest` folder, here is the command you would type in to convert the `splash.wav` file:

```
wav2gba splash.wav splash.bin
```

This will create a file called `splash.bin` in the current folder. Unfortunately for us, the `splash.bin` file is not exactly in the most useful format. What we need instead is a `splash.c` file with a byte array containing the `splash.wav` sample. The `splash.bin` file could be linked into the program via an assembler file or converted into an `.elf` file and linked into the `.exe`, but that is a lot more difficult than simply using a source code file (although I know some would not agree with me on that point). So, what is needed is a program to convert a raw binary file into a generic C source file containing an array of bytes. There is a program that is perfect for the job, called `bin2c.exe`, also written by Dark Fader. The syntax of this program is also very simple:

```
bin2c <input.bin> <output.c>
```

To convert the `splash.bin` file, you can simply use the `bin2c` program like this:

```
bin2c splash.bin splash.c
```

The resulting file looks like this (truncated for space):

```
const unsigned char splash[] =
{
0x00,0x00,0x00,0x00,0x00,0x00,0x7D,0x00,0x00,0x00,0x00,0x00,
0x18,0x2D,0x00,0x00,0xFA,0x00,0xFD,0xFD,0x03,0x00,0xFD,0x03,
0xFD,0x03,0x00,0xFA,0xFD,0xFD,0xFA,0xFD,0x03,0x00,0x00,0x03,
0x03,0x0C,0x00,0x06,0x00,0x03,0x03,0x00,0x00,0xFD,0xFA,0xFD,
. . .
0xFA,0xFD,0x00,0xFA,0x06,0x00,0xFD,0xFD,0xFD,0xF7,0x00,0xFD,
0xFD,0x03,0xFD,0xFA,0x03,0xFA,0x03,0x00,0xFD,0x03,0x09,0xFD,
0x00,0x00,0xFA,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
};
```

Since I use these two programs to convert waves so often, I wrote a short batch file, called `wav.bat`, that will convert a wave file to a C file, like this. You can use Notepad to write this short batch program:

```
@echo off
wav2gba %1.wav %1.bin
if exist %1.bin bin2c %1.bin %1.c
```

Now, instead of calling on two programs with a total of four parameters (which is a lot of typing when you need to convert a bunch of waves!), I simply type this:

```
wav splash
```

The batch file calls on `wav2gba` and `bin2c` to convert the wave to a C file. Simple!

I should mention something about conversion errors you are likely to encounter, some of which are obscure. There is one error that reads like this:

```
'data' not found
```

Another error message looks like this:

```
8 bit required
```

Both error messages are related to an unsupported wave file format. The files must be saved in a PCM wave format, so just load up a wave you are having trouble with into Cool Edit 2000 or a similar sound-editing tool, and then do a Save As to convert it to a PCM wave. That should take care of the problem. In some cases, you may also need to downsample the wave from 16 bits to 8 bits, because the `wav2gba` program is smart enough to know that only 8-bit samples will work on the GBA and will refuse to convert 16-bit samples. You will need to downsample those files. In Cool Edit 2000, you can do this from the Edit menu by selecting Convert Sample Type, or by simply pressing F11 to convert the wave.

Once you have converted a wave file, you need to keep the intermediate splash.bin file handy because it is a raw binary file, and you'll need to know the exact file size in bytes in order to plug that value into the SoundTest program. If you are still in the Command Prompt window from converting the file, you can get a list of files by typing "DIR" and taking note of the size of splash.bin. Otherwise, you'll have to right-click on splash.bin in Windows Explorer and click on Properties to get the exact file size in bytes. Just look at the Size, not the Size On Disk. Take a look at Figure 9.3. For reference, I noted that the file length is 11,568 bytes.

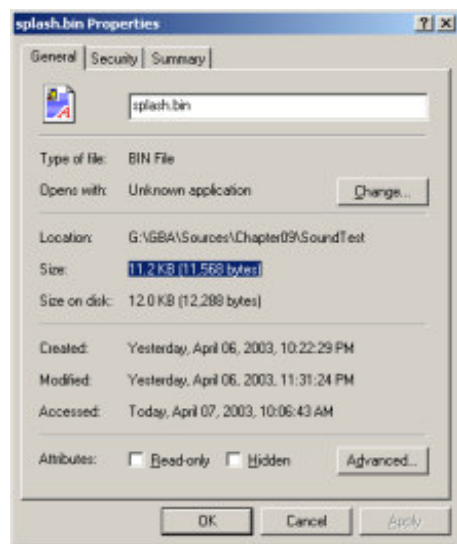


Figure 9.3

*The splash.bin file is 11,568 bytes in length, which is the exact length of the sound sample needed for the SoundTest program.*

## The SoundTest Header File

If you have successfully converted the splash.wav file (or if you merely looked in the SoundTest folder and found that it has already been converted!), then you are ready for the source code to SoundTest. This program is fairly short, so I recommend that you type it into Visual HAM—this is akin to getting your hands greasy by working on an engine, as opposed to hiring someone else to repair your car. You learn a great deal by doing it yourself!

If you are writing this program yourself, you will need to create a new project in Visual HAM called SoundTest. Add a new file by selecting File, New, New File. Be sure to select the radio button Add To Project and click on the C Header icon on the left. I have called the file main.h, but you may call it whatever you like, as long as you include this file in the main.c file. The dialog box is shown in Figure 9.4.

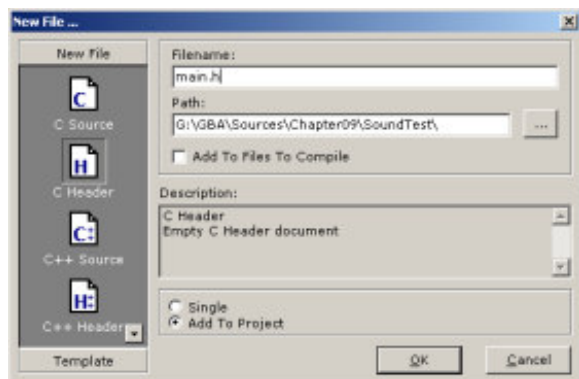


Figure 9.4

*The New File dialog box in Visual HAM.*

Now type the following code into the new main.h file. This code includes all the definitions needed to program the sound system on the GBA, including access to the DMA, timers, and interrupts needed to control sample playback.

```
////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 9: The Sound System
// SoundTest Project
// main.h header file
////////////////////////////////////

typedef unsigned char u8;
typedef unsigned short u16;
typedef unsigned long u32;

//define some video registers/values
#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
#define INT_VBLANK  0x0001

//define some timer and DMA registers/values
#define REG_TM0D      *(volatile u16*)0x4000100
#define REG_TM0CNT    *(volatile u16*)0x4000102
#define REG_DMA1SAD   *(volatile u32*)0x40000BC
#define REG_DMA1DAD   *(volatile u32*)0x40000C0
#define REG_DMA1CNT_H *(volatile u16*)0x40000C6
#define TIMER_ENABLE  0x80
#define DMA_DEST_FIXED 64
#define DMA_REPEAT    512
#define DMA_32        1024
#define DMA_ENABLE    32768
#define DMA_TIMING_SYNC_TO_DISPLAY 4096 | 8192

//define some sound hardware registers/values
```



```

#define REG_SGCNT0_H *(volatile u16*)0x4000082
#define REG_SGCNT1 *(volatile u16*)0x4000084
#define DSOUND_A_RIGHT_CHANNEL 256
#define DSOUND_A_LEFT_CHANNEL 512
#define DSOUND_A_FIFO_RESET 2048
#define SOUND_MASTER_ENABLE 128

```

## The SoundTest Source File

Now for the main source code file of SoundTest. This code should be typed into the main.c file (replacing the default code added by Visual HAM when the project was created).

```

////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 9: The Sound System
// SoundTest Project
// main.c source code file
////////////////////////////////////

#define MULTIBOOT int __gba_multiboot;
MULTIBOOT

#include "main.h"
#include "splash.c"

//global variables
u16 len = 0;

////////////////////////////////////
// Function: MyHandler
// Custom interrupt callback function
////////////////////////////////////
void MyHandler(void)
{
    //disable/store interrupts
    REG_IME = 0x00;
    u16 Int_Flag = REG_IF;

    //look for vertical refresh
    if((REG_IF & INT_VBLANK) == INT_VBLANK)
    {
        if (!len--)

```

```

    {
        //stop playback: disable the timer and DMA
        REG_TM0CNT = 0;
        REG_DMA1CNT_H = 0;
    }
}

//restore/enable interrupts
REG_IF = Int_Flag;
REG_IME = 0x01;
}

////////////////////////////////////
// Function: main()
// Entry point for the program
////////////////////////////////////
int main(void)
{
    u16 samplerate = 8000;
    u16 samplelen = 11568;
    u16 samples;

    SetMode(3 | BG2_ENABLE);

    //create custom interrupt handler for vblank (chapter 8)
    REG_IME = 0x00;
    REG_INTERRUPT = (u32)MyHandler;
    REG_IE |= INT_VBLANK;
    REG_DISPSTAT |= 0x08;
    REG_IME = 0x01;

    //output to both channels and reset the FIFO
    REG_SGCNT0_H = DSOUND_A_RIGHT_CHANNEL |
        DSOUND_A_LEFT_CHANNEL | DSOUND_A_FIFO_RESET;

    //enable all sound
    REG_SGCNT1 = SOUND_MASTER_ENABLE;

    //DMA1 source address
    REG_DMA1SAD = (u32)splash;

    //DMA1 destination address

```

```

REG_DMA1DAD = 0x40000A0;

//write 32 bits into destination every vblank
REG_DMA1CNT_H = DMA_DEST_FIXED | DMA_REPEAT | DMA_32 |
    DMA_TIMING_SYNC_TO_DISPLAY | DMA_ENABLE;

//set the sample rate
samples = 16777216 / samplerate;
REG_TM0D = 65536 - samples;

//determine length of playback in vblanks
len = samplelen / samples * 15.57;

//enable the timer
REG_TM0CNT = TIMER_ENABLE;

//run forever
while(1);
return 0;
}

```

Now that you have managed to get a sound to play through VisualBoyAdvance, I encourage you to create your own wave file and try to get it to play in the SoundTest program. The experience will be a valuable lesson in the process of converting a sound file, something you will end up doing often while working on a real game. Just remember, if you get any errors while trying to convert a wave file, you'll need to load it into a sound-editing program to downsample it to 8 bits, and you may also need to convert the wave to PCM. Make sure you are able to do this before moving on in the chapter.

## The PlaySamples Program

The PlaySamples program is an interesting program that demonstrates how to handle multiple sounds on the GBA. While I would love to get into mixing, as I mentioned before, it is too difficult of a subject to cover here. Even if I were to develop a sound mixer with you in this chapter, it would not be optimized. There are prebuilt sound libraries for the GBA, most of which are written entirely in ARM7 assembly language, that are extremely efficient. Several open source and freeware libraries are available, as are professional ones like Krawall. Again, you can select a library that is suitable for your needs by perusing the Web sites listed in Appendix B.

The PlaySamples program is shown in Figure 9.5. This program demonstrates not only how to handle multiple sounds but also how to keep track of the current position within the sample as it is being played. The numbers shown in the figure are not bytes but rather are the number of samples played per vblank period.

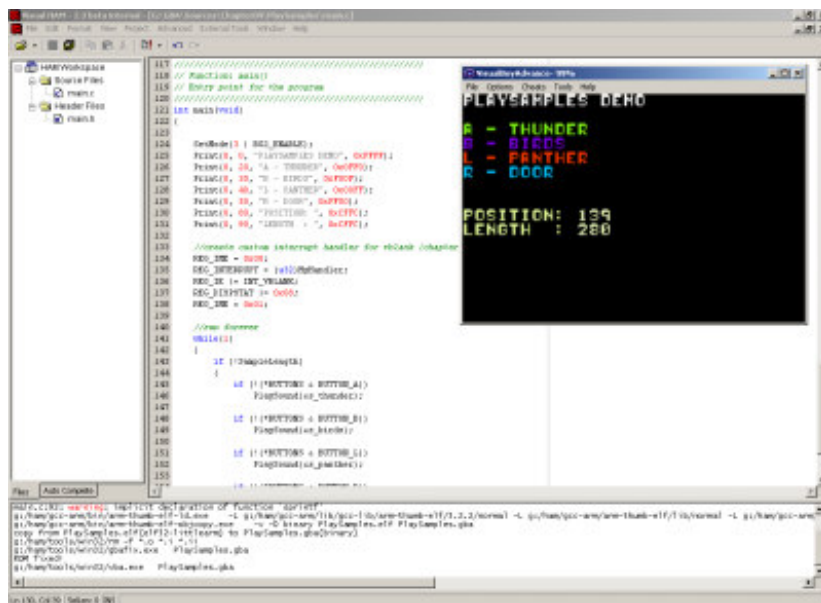


Figure 9.5  
The PlaySamples program demonstrates how to keep track of several sounds in a program, as well as how to track the current playback position.

## Tracking Sample Playback

The key to keeping track of the current playback position are two global variables, SampleLength and SamplePosition. I am one of the first programmers to worry about using global variables in this manner, but as I have mentioned several times in the past, it is sometimes better to go with the brute force approach in console development. While SamplePosition is just set to 0, SampleLength is a bit more than just the byte count. It is actually the number of sample groups processed by the DAC at a timed interval specified by a timer. The calculation I used, which compensates for the CPU cycles per second, works out to two lines of code:

```
samples = 16777216 / samplerate;
SampleLength = samplelength / samples * 15.57;
```

The 15.57 simply compensates for the timer and would have been better without the decimal, but this is just setup code, so speed isn't critical.

## The PlaySound Function

To facilitate the handling of multiple sound samples, I have converted the playback code from SoundTest into a reusable PlaySound function. Here is the complete function (which should look familiar to you after typing in the SoundTest program):

```
void PlaySound(sound *theSound)
{
    u16 samples;
```



```

//output to both channels and reset the FIFO
REG_SGCNT0_H = DSOUND_A_RIGHT_CHANNEL |
DSOUND_A_LEFT_CHANNEL | DSOUND_A_FIFO_RESET;

//enable all sound
REG_SGCNT1 = SOUND_MASTER_ENABLE;

//DMA1 source address
REG_DMA1SAD = (u32)theSound->pBuffer;

//DMA1 destination address
REG_DMA1DAD = 0x40000A0;

//write 32 bits into destination every vblank
REG_DMA1CNT_H = DMA_DEST_FIXED | DMA_REPEAT | DMA_32 |
    DMA_TIMING_SYNC_TO_DISPLAY | DMA_ENABLE;

//set the sample rate
samples = 16777216 / theSound->samplerate; //2097
REG_TM0D = 65536 - samples;

//keep track of the playback position and length
SampleLength = theSound->length / samples * 15.57;
SamplePosition = 0;

//enable the timer
REG_TM0CNT = TIMER_ENABLE;
}

```

## Keeping Track of Sounds

You might have noticed that the `PlaySound` function had a `sound` parameter instead of a `void*` pointer to a sound buffer. The `sound` struct helps to keep track of samples used in the program, so there aren't just a bunch of arrays or global variables (or at least, there are as few as possible). Here is what the `sound` struct looks like:

```

typedef struct tagSound
{
    void *pBuffer;
    u16 samplerate;
    u32 length;
}sound;

```

Now I'm skipping over the `#include` statements that load the actual sounds into the program. The sample arrays used in the `PlaySamples` program are called `panther`, `thunder`, `door`, and `birds`. Here is how I created the structs to keep track of these sounds. Note that each initialization also includes the sample's rate and length (which you must specify yourself, since the `bin2c` program didn't provide these values).

```
sound s_panther = {&panther, 8000, 16288};
sound s_thunder = {&thunder, 8000, 37952};
sound s_door = {&door, 8000, 16752};
sound s_birds = {&birds, 8000, 29280};
```

Simply calling `PlaySound` with one of these sound variables (`s_panther`, `s_thunder`, `s_door`, or `s_birds`) will start playback of that particular sample. The sample length value helps to determine when the sound output should be halted; this is done inside the interrupt handler for `vblank`. Without listing the entire interrupt callback function, here's the key code that takes care of shutting down the sound output when playback has reached the end of the sample:

```
SamplePosition++;
if (SamplePosition > SampleLength)
{
    REG_TM0CNT = 0;
    REG_DMA1CNT_H = 0;
    SampleLength = 0;
}
```

Each time through the `vblank` interrupt, a check is made to determine if `SamplePosition` is greater than `SampleLength`. The sound is actually halted by turning off the timer and also the DMA controller, both of which are responsible for providing new bytes to the DAC. Obviously, `SamplePosition` and `SampleLength` are globals, so this code can handle just one sample at a time. There is no means to stop a sample during playback and then resume, but if you wanted to just stop playback of a sample—for instance, to play a different sample—then you could set both of these registers to 0 in a generic `StopSound` function. I elected to just set the values inside the interrupt, but a `StopSound` function would be useful in an actual game.

## The PlaySamples Header File

Now that the theory is behind you, are you ready to get started on the source code for the `PlaySamples` program? It really is a simple program now that `PlaySound` has taken care of the details of setting up the sound registers and so on. So as in most cases, once the nitty-gritty is stuffed away in a reusable function, you can get down to business with the core program.

Okay, let's fire up Visual HAM and create a new project called PlaySamples. This program requires the font.h file, which you may copy from an earlier project, such as the Framerate program in the previous chapter. Add a new file to the PlaySamples project called main.h, and type in the following code:

```
////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 9: The Sound System
// PlaySamples Project
// main.h header file
////////////////////////////////////

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

//function prototypes
void Print(int left, int top, char *str, unsigned short color);
void DrawChar(int left, int top, char letter, unsigned short color);
void DrawPixel3(int x, int y, unsigned short color);

//define some video registers/values
unsigned short* videoBuffer = (unsigned short*)0x6000000;
#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
#define INT_VBLANK 0x0001
```

```

//define some timer and DMA registers/values
#define REG_TM0D          *(volatile u16*)0x4000100
#define REG_TM0CNT       *(volatile u16*)0x4000102
#define REG_DMA1SAD      *(volatile u32*)0x40000BC
#define REG_DMA1DAD      *(volatile u32*)0x40000C0
#define REG_DMA1CNT_H    *(volatile u16*)0x40000C6
#define TIMER_ENABLE     0x80
#define DMA_DEST_FIXED   64
#define DMA_REPEAT       512
#define DMA_32           1024
#define DMA_ENABLE       32768
#define DMA_TIMING_SYNC_TO_DISPLAY 4096 | 8192

//define some sound hardware registers/values
#define REG_SGCNT0_H     *(volatile u16*)0x4000082
#define REG_SGCNT1      *(volatile u16*)0x4000084
#define DSOUND_A_RIGHT_CHANNEL 256
#define DSOUND_A_LEFT_CHANNEL 512
#define DSOUND_A_FIFO_RESET 2048
#define SOUND_MASTER_ENABLE 128

//define button hardware register/values
volatile unsigned int *BUTTONS = (volatile unsigned int *)0x04000130;
#define BUTTON_A 1
#define BUTTON_B 2
#define BUTTON_R 256
#define BUTTON_L 512

////////////////////////////////////
// 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);
}

```

## The PlaySamples Source File

The main source code for the PlaySamples program should be typed into the main.c file (and as usual, be sure to first delete the skeleton code added to the file by Visual HAM). Now here is the main code for the program:

```

////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 9: The Sound System
// PlaySamples Project
// main.c source code file
////////////////////////////////////

#define MULTIBOOT int __gba_multiboot;
MULTIBOOT

#include "main.h"
#include "panther.c"
#include "thunder.c"
#include "door.c"
#include "birds.c"

//create a struct to keep track of sound data
typedef struct tagSound
{
    void *pBuffer;
    u16 samplerate;
    u32 length;
}sound;

//create variables that describe the sounds
sound s_panther = {&panther, 8000, 16288};
sound s_thunder = {&thunder, 8000, 37952};
sound s_door = {&door, 8000, 16752};

```

```

sound s_birds = {&birds, 8000, 29280};

//global variables
u16 SamplePosition = 0;
u16 SampleLength = 0;
char temp[20];

////////////////////////////////////
// Function: PlaySound
// Plays a sound sample using DMA
////////////////////////////////////
void PlaySound(sound *theSound)
{
    u16 samples;

    //output to both channels and reset the FIFO
    REG_SGCNT0_H = DSOUND_A_RIGHT_CHANNEL |
        DSOUND_A_LEFT_CHANNEL | DSOUND_A_FIFO_RESET;

    //enable all sound
    REG_SGCNT1 = SOUND_MASTER_ENABLE;

    //DMA1 source address
    REG_DMA1SAD = (u32)theSound->pBuffer;

    //DMA1 destination address
    REG_DMA1DAD = 0x40000A0;

    //write 32 bits into destination every vblank
    REG_DMA1CNT_H = DMA_DEST_FIXED | DMA_REPEAT | DMA_32 |
        DMA_TIMING_SYNC_TO_DISPLAY | DMA_ENABLE;

    //set the sample rate
    samples = 16777216 / theSound->samplerate; //2097
    REG_TM0D = 65536 - samples;

    //keep track of the playback position and length
    SampleLength = theSound->length / samples * 15.57;
    SamplePosition = 0;

    //enable the timer
    REG_TM0CNT = TIMER_ENABLE;

```

```

}

////////////////////////////////////
// Function: MyHandler
// Custom interrupt callback function
////////////////////////////////////
void MyHandler(void)
{
    u16 Int_Flag;

    //disable interrupts
    REG_IME = 0x00;

    //backup the interrupt flags
    Int_Flag = REG_IF;

    //look for vertical refresh
    if((REG_IF & INT_VBLANK) == INT_VBLANK)
    {
        //is a sample currently playing?
        if (SampleLength)
        {
            //display the current playback position
            DrawBox3(80, 80, 120, 100, 0x0000);
            sprintf(temp, "%i", SamplePosition);
            Print(80, 80, temp, 0xDFFD);
            sprintf(temp, "%i", SampleLength);
            Print(80, 90, temp, 0xDFFD);

            //increment the position, check if complete
            SamplePosition++;
            if (SamplePosition > SampleLength)
            {
                //stop playback: disable the timer and DMA
                REG_TM0CNT = 0;
                REG_DMA1CNT_H = 0;
                //reset length
                SampleLength = 0;
            }
        }
    }
}

```



```

    }

    //restore the interrupt flags
    REG_IF = Int_Flag;

    //enable interrupts
    REG_IME = 0x01;
}

////////////////////////////////////
// Function: main()
// Entry point for the program
////////////////////////////////////
int main(void)
{
    SetMode(3 | BG2_ENABLE);
    Print(0, 0, "PLAYSAMPLES DEMO", 0xFFFF);
    Print(0, 20, "A - THUNDER", 0x0FF0);
    Print(0, 30, "B - BIRDS", 0xF00F);
    Print(0, 40, "L - PANTHER", 0x00FF);
    Print(0, 50, "R - DOOR", 0xFF00);
    Print(0, 80, "POSITION: ", 0xCFFC);
    Print(0, 90, "LENGTH  :", 0xCFFC);

    //create custom interrupt handler for vblank (chapter 8)
    REG_IME = 0x00;
    REG_INTERRUPT = (u32)MyHandler;
    REG_IE |= INT_VBLANK;
    REG_DISPSTAT |= 0x08;
    REG_IME = 0x01;

    //run forever
    while(1)
    {
        if (!SampleLength)
        {
            if (!(*BUTTONS & BUTTON_A))
                PlaySound(&s_thunder);

            if (!(*BUTTONS & BUTTON_B))
                PlaySound(&s_birds);
        }
    }
}

```

```

        if (!(*BUTTONS & BUTTON_L))
            PlaySound(&s_panther);

        if (!(*BUTTONS & BUTTON_R))
            PlaySound(&s_door);
    }
}
return 0;
}

```

If all goes well, you should now have a simple but useful sound engine for your next game project. Sound effects are at least as important as the graphics in a game, so don't put sound on the side while working on the "more important" aspects of your next game. A well-designed game uses sound to greatly enhance the gaming experience.

## Summary

This chapter has been an overview of the sound hardware on the GBA. You learned about the Direct Sound A and Direct Sound B channels and how to create, convert, and play samples. This chapter provided a simple demonstration of playing a single sound, followed by a more useful program that was able to play one of several sound effects based on button presses. There is obviously more to sound programming than has been covered in this single chapter, but you now have enough information to add sound support to your GBA programs. For more advanced sound capabilities, including the ability to play modules as music tracks in addition to sound mixing, I recommended going with a sound library such as Krawall, since it is rare even among commercial GBA developers to write a custom sound library when excellent prebuilt solutions are already available for a small licensing fee.

## 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:** The SoundTest program is a simple and easy way to test converted wave files. See if you can convert your own wave files with wav2gba and bin2c, as explained in this chapter, to gain some experience converting and playing sound files.


**Challenge 2:** The PlaySamples program displays the playback position in sample blocks per vblank. Modify the program so it shows both the position and length of the sample in actual bytes.

**Challenge 3:** The PlaySamples program supports just four sounds, using the A, B, L, and R buttons. Enhance the program by adding more sounds and make use of the other buttons: Up, Down, Left, Right, Start, and Select.

# 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. How many *total* sound channels are built into the GBA sound system?
  - A. 2
  - B. 4
  - C. 6
  - D. 8
  
2. True or False: The GBA sound system supports stereo sound.
  - A. True
  - B. False
  
3. What are the two digital sound channels called?
  - A. Frequency and Modulation
  - B. Digital Sound 1 and Digital Sound 2
  - C. Direct Sound A and Direct Sound B
  - D. FM Synthesis and Wave Table
  
4. What utility program is used in this chapter to convert a wave file?
  - A. wav2c
  - B. wav2bin
  - C. bin2long
  - D. wav2gba
  
5. What sampling resolution is supported by the GBA's sound system?
  - A. 8-bit
  - B. 12-bit
  - C. 16-bit
  - D. 24-bit
  
6. What does it mean if a sound sample has a frequency of 44.1 kHz?
  - A. The sample will be played back quickly.
  - B. The sound has been undersampled.

- 
- C. The sample was recorded from a radio station.  
D. The sample contains 44,100 samples per second.
7. What is the name of the direct sound control register at memory address 0x04000082?
- A. REG\_SNDCNT
  - B. REG\_SOUND\_CNT\_L
  - C. REG\_SOUND\_CNT\_H
  - D. REG\_DS\_CNT
8. How many CPU cycles does the GBA execute per second?
- A. 32,768
  - B. 16,777,216
  - C. 1,024
  - D. 65,535
9. What wave file format does the GBA sound system support exclusively?
- A. PCM
  - B. A/mu-Law Wave
  - C. ACM Waveform
  - D. DVI/IMA ADPCM
10. What is the name of the sound mixing and ProTracker music playback library mentioned in this chapter?
- A. Tidal Wave
  - B. Cool Tunes
  - C. Kurzweil
  - D. Krawall