

Covers
Flash MX 2004

FLASH HACKS™

*100 Industrial-Strength
Tips & Tools*



O'REILLY®

Sham Bhangal

HACK
#52

Create a Flash Speech Synthesizer

Synthesize speech for added flexibility with reduced bandwidth—no need to record or download predetermined sounds.

This hack uses Flash sound events to splice sound samples seamlessly together to simulate one continuous sound effect, in this case digitized speech using *allophones* (phonetic speech sections from which the spoken word is built up).

The Trouble with Lasers

Some hacks rely on standard techniques or technologies used for an unexpected application. Science is filled with unexpected uses of technologies, such as lasers. Few envisioned that lasers could be used for everything from warfare to surgery, to listen to music, watch movies, and transmit international phone conversations.

In this hack, we'll see how the seemingly innocuous *onSoundComplete* event can be used to create a speech synthesizer in Flash.

Using Sound.onSoundComplete Event

As soon as the *Sound.onSoundComplete()* method came out in Flash MX, I thought, "Cool! We can now easily synchronize sound accurately enough to create a sound mixing board," and sure enough, we can. The *onSoundComplete* event is accurate to a resolution of around 10 times the frame rate, so it's far more accurate than the *keyframe/onEnterFrame*-based sound controls common in Flash 5.

Like all event-driven code, *onSoundComplete* events might be delayed if there is a lot of other stuff going on at the same time, particularly processor-intensive animation. Overtaxing the Flash Player may result in slight hitches (delays) in playback, so go easy!

Have a look at the following code (or have a look at the file *groovyLoop.fla* from this book's web site):

```
function nextSound () {  
    myLoop.start();  
}  
myLoop = new Sound(this);  
myLoop.attachSound("groovy");  
myLoop.start();  
myLoop.onSoundComplete = nextSound;
```

For this code to work, you need to have a sound file in the Library panel with the linkage identifier *groovy*. Although this is a standard requirement

when using scripted sound, it is also the most likely cause of your sounds not playing if you get it wrong. Let's go through it:

1. After importing your sound file into Flash (with File → Import → Import to Library), right-click (Windows) or ⌘-click (Mac) on the sound file asset in the Library. Select Linkage from the contextual menu that appears.
2. In the Linkage Properties dialog box that appears, as shown in [Figure 7-1](#), enter the linkage identifier **groovy** in the Identifier field. This can be any text string you want, but don't make it too complicated, long, or unmemorable, because you need to use exactly the same linkage identifier (surrounded by quotes) in your code.
3. Check the Export for ActionScript checkbox. The Export in First Frame checkbox will become selectable and checked when you do this. Leave Export in First Frame checked.

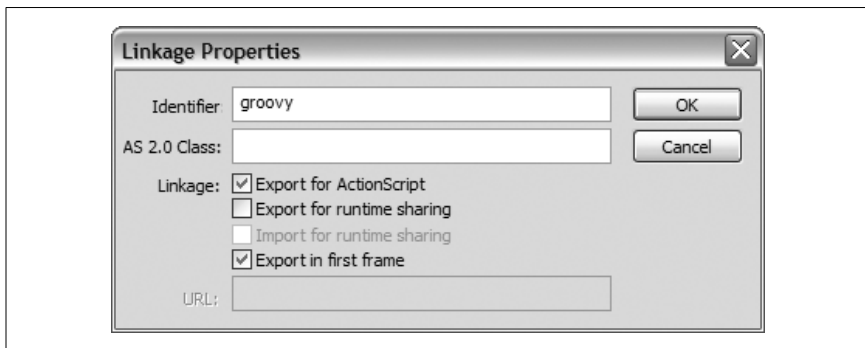


Figure 7-1. The Linkage Properties window, correctly filled in to define the groovy linkage identifier

As part of the process of compiling a SWF, Flash searches through your timelines to see when all your symbols are attached. It orders content in the SWF based on this. Thus, when the SWF is streamed over the Web, the symbols are loaded into the Flash Player in the order they are required. For symbols that are attached to timelines via scripting only, Flash will not see the symbols on the timeline during SWF compilation and does not add them to the SWF by default, in the assumption that they are unused. Checking the Export for ActionScript and Export in First Frame checkboxes overrides this default, and in doing so, you are telling Flash, “I will be using this symbol via scripting, so place the symbol in the SWF.”

However, this forces the Flash Player to download the sound before frame 1 is even displayed. A better approach is to place your sound in a movie clip and use the `Sound.stop()` command to prevent it from playing immediately

[Hack #55]. Then place that movie clip on the frame of the main timeline in which you want the sound exported (i.e., the frame before you need the sound). But sound preloaders are not the subject of this hack, so let's return to our previous setup.

In the earlier *groovyLoop.fla* listing, a sound with the linkage identifier *groovy* is attached to the *Sound* instance *myLoop*. When the current sound completes playing, the *onSoundComplete* event invokes the *nextSound()* function, which sets the sound going again. Well, so what? Surely something like the following can accomplish the same thing:

```
myLoop = new Sound(this);  
myLoop.attachSound("groovy");  
myLoop.start(0, 1000);
```

Here we are using the *Sound.start()* method to cause our *groovy* sound to repeat 1000 times, more than long enough for most people to agree that our sound goes on forever.

The magic of *onSoundComplete* is that it is dynamic; you can change the sound on the fly to create things like sound mixing boards, because the callback function (in this case *nextSound()*) can choose a different sound to start at the end of the current one. You can thus create dynamic sound tracks on which the sound loops are changed interactively or based on other events. We won't look too deeply at that route here because it's a standard usage and not part of the hack. Instead, let's create a hack using a novel application that is fundamentally the same process.

Speech and Flash

When I first learned about *onSoundComplete*, I was also looking at accessibility and screen readers (plugins that the visually impaired use to convert web content into an audio speech stream). They are really cool, but they weigh in at more than 10 MB. Way back when we were all playing around with Commodore 64s and Atari consoles, speech could be created in about 32 KB. The trick was using a set of individual phonetic sounds called allophones, such as the “ou” in “you” and the “th” in “the.” I got to thinking that the code that created speech in the old 64 KB computers must have spliced all these tiny allophones together using something like *onSoundComplete*. More to the point, I was pretty sure it shouldn't take 10 MB to do it, and that was the thing driving me to create this hack.

Although few people have the old 8-bit computer hardware lying around anymore, you can find numerous software emulators. Searching for “8-bit emulation” or “8-bit emulation allophone” in my search engine of choice (<http://vivisimo.com>) brings up a large number of emulator home pages.

Many of these emulators are capable of emulating speech synthesizer hardware that was available for the emulated systems.

Fonetikally Speaking

Importing some allophones into Flash was a breeze. I used Adobe Audition (a sound-editing application) to:

- Remove any silent spaces at the beginning and end of each sample
- Reduce any clicks at the start and end of the samples (using fade in or fade out if necessary)
- Normalize the samples (so that all samples are at the same volume and power level)
- Optimize the samples for web use

I ended up with a library of synthesized voice samples, each of which I gave a name and linkage identifier that were both identical to the spoken allophone, as shown in [Figure 7-2](#). For those not prepared to go through the same process, you can download *speech_final fla*, complete with allophone sounds already imported, from this book's web site.

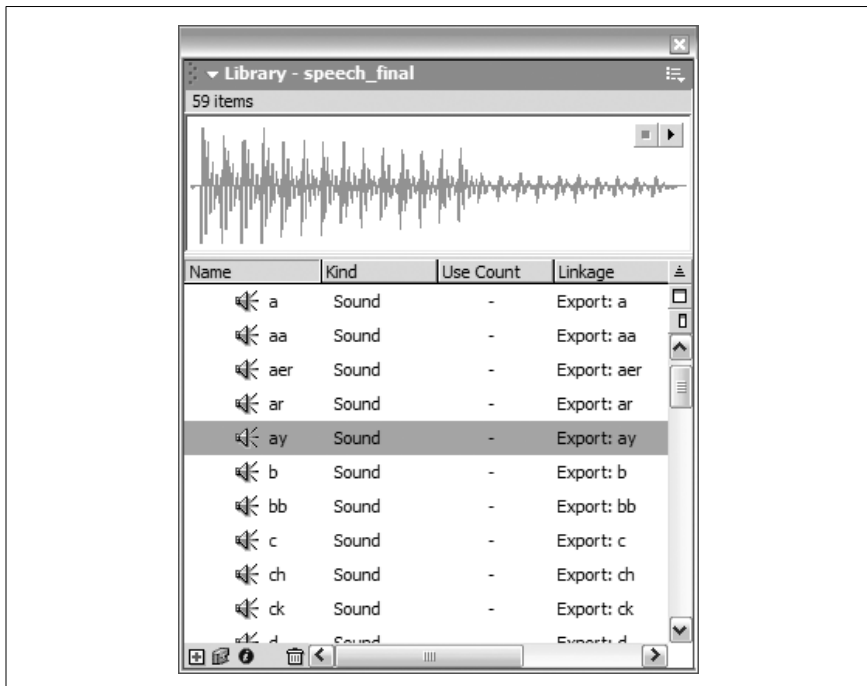


Figure 7-2. The Library, showing the sound files used for the allophones

The old 8-bit speech synthesizers did not convert conventional spelling to phonetic text. For example, to pronounce the word “knowledge,” you had to ask the speech synthesizer to pronounce the phonetic equivalent, such as “nnolej” or something similar. To test my premise that Flash can be made to talk, I limited myself to writing a phonetic speech synthesizer. If you want to make a name for yourself, I’d welcome any code that converts from conventional spelling to phonetic spelling. (You might begin with technical linguistics papers you can find on the Web!)

So I tacked some string handling onto the simple sound code shown previously to create something that could take a raw allophone-based string and convert it to a simple sound queue serviced by an *onSoundComplete()* callback function:

```
makePhrase = function () {
    if (soundCount < soundMax) {
        soundCount++;
        speech.attachSound(aPhones[soundCount]);
        speech.start();
    } else {
        delete speech.onSoundComplete;
    }
};

function say (phrase) {
    var i = j= 0;
    aPhones = new Array();
    for (i = 0; i < phrase.length; i++) {
        if (phrase.charAt(i) != "|") {
            aPhones[j] += phrase.charAt(i);
            if (phrase.charAt(i) == " ") {
                aPhones[j] = "space";
            }
        } else {
            j++;
        }
    }
    speech.attachSound(aPhones[0]);
    speech.start();
    speech.onSoundComplete = makePhrase;
    soundCount = 0;
    soundMax = j-1;
}

function SoundInit () {
    speech = new Sound(this);
}

soundInit();
say("h|e|ll|oo| | | | |h|ow| |ar| |y|ouu| | | |tt|u|d|ay|");
stop();
```

Unfortunately, some of the allophone sound names are too similar to differentiate between them easily. For example, “ooo” could be written as “o o o”, “o oo”, or “oo o” and each variation would sound different. Instead, I used the | character as an allophone separator. The pure programmer in me would have renamed the allophones to “o1” and “o2” so the problem never occurs, but the designer in me overturned this decision because it makes manually creating the phonetic text string easier and more intuitive—“y|oo” makes more sense than “y|o2.”

Assuming we just wanted to say “hello,” the code works as follows. The first line sets the ball rolling. It sends our text to be spoken as a phonetically spelled string argument to the `say()` function:

```
say("h|e|ll|oo|");
```

The `say()` function creates a new array, which overwrites any existing array of the same name, so we don’t need to clear data from previous function calls. Following our example through, the array of allophones is:

```
aPhones[0] = "h"  
aPhones[1] = "e"  
aPhones[2] = "ll"  
aPhones[3] = "oo"
```

We then attach the first sound, “h”, to our *Sound* instance and use *onSoundComplete* events to sequentially attach “e”, “ll”, and “oo” in turn as each allophone completes. This is taken care of via the *makePhrase()* function, which attaches the next sound file in our allophone sequence, `aPhones`, until all the entries in the array are sounded. Simple, eh? It works in a cool retro Speak & Spell digitized voice. Have a listen to *speech_final fla*.

As a final hack-within-a-hack, note that I have used only one *Sound* instance, rather than one per sound file, which results in longer code full of too many *for* loops. Using one *Sound* instance is possible because you can use *Sound.attachSound()* as many times as you like, and it is also fast, so you can do it on the fly. This has two very cool side effects for the advanced Flash sound programmer:

- You need to set up only one *onSoundComplete* callback rather than having to create one per *Sound* instance.
- Flash has eight sound channels, but you cannot normally access the separate channels. Flash assigns the channels behind the scenes, but this is not always desirable if you are creating complex scripted sound control. Instead, you can create eight *Sound* instances (`channel0` to `channel7`), and then reassign the sounds to be played dynamically to each via *Sound.attachSound()*. This gives you your eight *Sound* instances in a way that relates directly to the eight sound channels.

Final Thoughts

A Flash-based speech synthesizer is so cool because it carries its own speech engine in the SWF itself; no accessibility plugins are required. Although commercial screen readers such as GW Micro's Windows-Eyes (<http://www.gwmicro.com>) are available, they require the *Accessibility* class, which requires Windows ActiveX. The Flash speech code works happily on any system that supports the Flash Player's full sound capabilities.

You can build something that doesn't even require the user to look at the screen. (I picture someone carrying around a palmtop that says, "Reminder: your dental appointment is at 10 today.") You could also add speech to devices that have small displays so that things like help text don't clutter the valuable screen real estate.

Not just useful for sight-impaired users, it's also a new Flash-to-user info stream for everyone else. How many new applications does that put on the horizon?

Aren't lasers wonderful!