Java Sound, Compressing Audio with mu-Law Encoding

Baldwin shows you how to use mu-law encoding and decoding to compress and restore 16-bit linear PCM samples.

Published: December 2, 2003
By Richard G. Baldwin

Java Programming Notes # 2026

Preface

This series of lessons is designed to teach you how to use the Java Sound API. The first lesson in the series was entitled Java Sound, An Introduction. The previous lesson was entitled Java Sound, Audio File Conversion.

Two types of audio data

Two different types of audio data are supported by the Java Sound API:

- Sampled audio data
- Musical Instrument Digital Interface (MIDI) data

The two types of audio data are very different. I am concentrating on sampled audio data at this point in time. I will defer my discussion of MIDI until later.

Viewing tip

You may find it useful to open another copy of this lesson in a separate browser window. That will make it easier for you to scroll back and forth among the different listings and figures while you are reading about them.

Supplementary material

I recommend that you also study the other lessons in my extensive collection of online Java tutorials. You will find those lessons published at Gamelan. COM. However, as of the date of this writing, Gamelan doesn't maintain a consolidated index of my Java tutorial lessons, and sometimes they are difficult to locate there. You will find a consolidated index at www.DickBaldwin.com.

Material in earlier lessons

Earlier lessons in this series showed you how to:

- Perform file conversions among different audio file types.
- Create, play, and save synthetic sounds, making use of the features of the java.nio package to help with the byte manipulations.
- Use methods of the **AudioSystem** class to write more robust audio programs.
- Play back audio files, including those that you create using a Java program, and those that you acquire from other sources.
- Capture microphone data into audio files types of your own choosing.
- Capture microphone data into a **ByteArrayOutputStream** object.
- Use the Sound API to play back previously captured audio data.
- Identify the mixers available on your system.
- Specify a particular mixer for use in the acquisition of audio data from a microphone.
- Understand the use of lines and mixers in the Java Sound API.

This lesson will show you how to use mu-law encoding and decoding to compress and restore 16-bit linear PCM samples.

General Discussion of Encoding and Compression

Why compress?

With the advent of the Internet and the desire to share pictures, music, movies, and other items involving voluminous amounts of data, the need for data compression has become evident to the general public.

You will find some very interesting information about sampled sound published by Marc Boots-Ebenfield at Sound Formats. Included on this web site is the following factoid regarding CD quality music.

> *"On a Music CD the music is sampled at 44.1 KHz using 16 bit words or 705,600 bits for each second of sound. At 8 bits to the byte that would mean that 1 second of CD quality music would occupy 88,200 bytes or 88 Kb of your floppy disc which holds 1.2 Mb of data. That means that you could hold 13 seconds of CD quality music on a floppy- (uncompressed)!"*

Thus, if the above estimate is correct, about fifteen floppy disks would be required to contain a typical three-minute song in uncompressed CD quality format. However, long before the Internet existed, there was the need to perform data compression in order to reduce the bandwidth requirements for transmitting data.

What is mu-law encoding?

There are many data compression schemes used to provide bandwidth reduction for the storage or transmission of sound. One of the schemes supported by the Java Sound API is a data encoding scheme commonly known as *mu-law, u-law, ulaw, ULAW,* or something similar.

My preparation effort for writing this article has proven to be an interesting exercise in gleaning information from the Internet. My objective was to provide you with an understanding of mu-law encoding and decoding.

Needed to fill in some technical gaps

While I had a pretty good idea of what mu-law encoding was all about at the outset, there were some technical gaps in my knowledge that needed to be filled. A Google search on the words in the above list produced thousands of hits. However, it was surprisingly difficult to gather enough definitive information to fully understand how mu-law encoding actually works, and to be able to write understandable programs to perform mu-law encoding and decoding using the Java programming language.

Many sources, little definitive information

I consulted many different Internet sources to gain a good understanding of mu-law encoding before tackling this lesson. With respect to actual programming, two of the best sources that I found are accessible at link1 and link2.

> *(I will refer back to these links at several points in this document, so you might want to bookmark them at this point. Note that the second link is a Word document. If, like myself, you are not comfortable downloading and opening a Word document, do a Google search for the keywords* ***David Callier mu-law****. That should lead you to an HTML version of his semester project report entitled* ***mu-Law Speech Compression****, which should be safe to download.)*

C-language programs

The first link listed above provides mu-law encoding and decoding programs written in what appears to be the C language.

> *(These programs make very heavy use of table lookups with no explanation as to the source of the values in the tables. Therefore, the*

programs aren't very useful from an understanding viewpoint. However, they are useful from an algorithm validation viewpoint, since it is relatively easy to convert the programs to the Java programming language and run them without understanding exactly why they do what they do.)

Encoding and decoding formulas

The second link presents the encoding and decoding algorithms in formulas, and provides a reasonable explanation of those formulas. Although it appears to provide some programming as well, I didn't recognize the programming language being used.

(Again, the explanations in this resource involve the manipulation of data using constants with no explanation of the meaning of the constants.)

Combine and conquer

I was able to combine the information from these two sites, along with information from many other sites, to write Java code that successfully performs mu-law encoding and decoding.

Why are there so many names for the same thing?

Now let's backtrack to the list of names that I gave you earlier. Apparently, the original name given to this encoding scheme consisted of the Greek letter **mu** *(which looks a lot like a lower-case **u** in the English alphabet)* and the word **law**. Because of the visual similarity between **mu** and **u**, and because many people including myself don't know how to cause the Greek letter **mu** to be properly rendered in an HTML page, numerous variations on the name have evolved, as shown in the above list.

A description of mu-law encoding

Here is how one Internet author describes mu-law encoding:

"Mu-law (also "u-law") encoding is a form of logarithmic quantization or companding. It's based on the observation that many signals are statistically more likely to be near a low signal level than a high signal level. Therefore, it makes more sense to have more quantization points near a low level than a high level. In a typical mu-law system, linear samples of 14 to 16 bits are companded to 8 bits. Most telephone quality codecs (including the Sparcstation's audio codec) use mu-law encoded samples. "

What is companding?

Here is a paraphrase of how another author describes *companding.* He also provides a mathematical definition of mu-law format as well:

"A *companding* operation *compresses* dynamic range on encode and *expands* dynamic range on decode. In digital telephone networks and voice modems (currently in use everywhere), standard *CODEC*[C.9]chips are used in which audio is digitized in a simple 8-bit _*-law format* (or simply ``mu-law").

Given an input sample _ represented in some internal format, such as a `short`, it is converted to 8-bit mu-law format by the formula [48]

_

where _ is a *quantizer* which produces a kind of logarithmic fixed-point number with a 3-bit characteristic and a 4-bit mantissa, using a small table lookup for the mantissa. As we all know from talking on the telephone, mu-law sounds really quite good for voice, at least as far as *intelligibility* is concerned*. ..."*

As you can see, the above quotation also contains numerous hyperlinks that will take you to other pages that contain related information. If some of those links become broken over time, just refer to the link for the original author.

Data compression in general

We compress audio data because we want to reduce the amount of storage space required to store the data, or reduce the bandwidth required to move the data across a network.

In theory, if we acquire perfectly quantized samples of an analog signal at a uniform rate which is at least twice has high as the highest frequency component in the analog signal being sampled, we can use a perfect digital-to-analog converter to perfectly reconstruct the analog signal from the samples. In that case, a comparison of the reconstructed signal with the original analog signal would produce zero error.

The world is not perfect

You may have noticed the use of the word *perfect* several times in the above paragraph. Unfortunately, there are no perfect systems. However, it is often possible to achieve results that, while not perfect, are adequate for the intended purpose.

In order to reduce the bandwidth or storage requirements associated with a given sampled analog signal, we must reduce the amount of digital data required to describe that analog signal. Generally, this is accomplished in either or both of two ways:

- Reduce the number of samples used to describe the analog signal.
- Reduce the number of bits used to describe each sample.

What is a *bit* of information?

Despite the fact that the widespread use of computers is a relatively new phenomenon, significant work has been going on in information theory since well before the Second World War. I believe that it was a scientist named Shannon who first described one *bit* of information. His description was something like the following:

> *One bit of information is the amount of information required to determine the outcome for a process in which there are two equally probable outcomes.*

Since then, we have corrupted the use of the term *bit,* using it as the term that describes the smallest unit of information in a digital system, without regard to probabilities. However, it is useful to think in terms of probabilities when thinking about how to reduce data storage and transmission bandwidth requirements.

Systems with redundancies

Many systems that describe analog signals using digital data contain redundant data bits. In other words, each data bit used to describe the analog signal is not charged with deciding between two equally probable outcomes *(as Shannon described the purpose of one bit of information).* This fact can often be used to advantage in order to implement *lossless* data acquisition and encoding schemes that reduce storage and bandwidth requirements.

Common zip files

One form of lossless data compression familiar to most of us is the common zip file. The process of encapsulating data into a zip file uses a mathematical algorithm to reduce or eliminate redundancies in order to represent the same data in a smaller file. Some files, such as text files, contain significant redundancies, resulting in compression factors as great as fifty to sixty percent. Other files contain less redundancy, and hence result in less compression when encapsulated in a zip file. *(To demonstrate this latter situation, encapsulate a zip file into a zip file. You will see very little, if any further compression.)*

Assuming that the zip file is not corrupted, the mathematical algorithm used to compress the data is completely reversible. Therefore, you can extract a file from a zip file with no loss of information. Therefore, this is a completely lossless compression process.

Reduction of sampling rate

Some other processes may not be mathematically lossless, but may be lossless from a practical viewpoint. For example, if a very low bandwidth analog signal is being sampled at a very high sampling rate, this produces redundant data. The amount of data required to describe such a low-bandwidth signal can often be reduced by reducing the sampling rate. As long as this is done with care, the samples taken at the lower sampling rate can be used just as effectively as the samples taken at the higher sampling rate for the purpose of reproducing the analog signal. No information is lost by virtue of taking samples at a lower rate, and hence this is a lossless system from a practical viewpoint *(as opposed to lossy system that I will describe later).*

Reduction of bits per sample

As another example of a lossless system, if a black and white photograph is scanned using a scanning algorithm that is designed to properly scan color photographs and to maintain information about millions of colors, the resulting file may contain redundant data bits.

It may be that a different scanning algorithm designed specifically for black and white photographs could do the job equally well while creating much smaller data files. The black and white algorithm might use fewer data bits to describe each sample than would be the case with the color algorithm, because there would be significantly fewer possibilities that would need to be described at each sample point for the black and white photograph.

Redundant data in a remote plotting system

As still another example of redundant data, many years ago, in the days of *remote batch* data processing using mainframe computers, I developed a lossless compression scheme for transmitting plotting data from mainframe computers to remote terminals that were driving digital pen plotters. This scheme was also based on reducing the number of bits in each data sample, and was used profitably by my company for many years.

Absolute coordinates versus change in coordinates

Typical computer systems that drove digital pen plotters in those days transferred a new pair of coordinate values to the plotting system each time it was necessary to move the pen. The coordinate values were typically quite large, and each coordinate value required several bytes of information to be transmitted.

However, the distance that the pen moved during each movement was typically quite small, and the coordinate values contained lots of redundant data bits. It was much more efficient to transmit the changes in coordinate values associated with each movement than to transmit the new coordinate values for each movement, so that is what we did. *(Actually the algorithm was designed to switch between the two modes, depending on which was most efficient at a particular point in the plotting process.)*

Lossy compression systems

A lossless compression system is possible only when there is redundancy in the data, which can be eliminated to advantage. Frequently, it is also advantageous to use *lossy* compression systems even when it is known that some signal degradation or information loss may occur *(obviously, you wouldn't want to use a lossy compression scheme with financial data, but it may be alright with audio data).*

Many data encoding and compression systems, including systems based on mu-law encoding, are lossy. This means that a compressed signal cannot be uncompressed without some loss of information. In situations such as this, the user must decide whether the benefits achieved through lossy encoding and compression outweigh the penalties associated with the loss of information.

MP3 encoding and compression

A good example of this is the use of MP3 encoding and compression for the recording and transmission of music files. I have no expertise in this area, but it is my understanding based on the reading that I have done that MP3 is a lossy encoding scheme. The quality of the uncompressed music signals following MP3 encoding and compression is somewhat degraded relative to the original CD-quality music. However, the quality is apparently good enough to satisfy millions of music fans who make heavy use of MP3.

Why use mu-law encoding for audio compression?

Although somewhat lossy, mu-law encoding and compression is apparently adequate for the storage and transmission of spoken voice signals in many cases. Thus, it is heavily used for digital telephone networks where it achieves a data compression factor of approximately two-to-one relative to 16-bit PCM data.

Mu-law encoding has a number of important characteristics, not the least of which is the ability to spread the information loss across a wide range of signal levels. In other words, on a percentage basis, the level of information loss experienced by low-level signals is approximately the same as the level of information loss experienced by high level signals.

Thus, mu-law encoding is capable of maintaining a relatively wide dynamic range and to spread the information loss across that entire range. This will be illustrated by the program that I will discuss later in this lesson.

Preview

In this lesson, I will teach you how to write Java programs to encode 16-bit linear PCM audio samples into 8-bit mu-law bytes, and how to decode 8-bit mu-law bytes back into 16-bit PCM samples.

I will perform an experiment to determine the distribution of the information loss across a dynamic range of approximately 96 decibels, and will compare that distribution of information loss with the information loss produced by simply truncating 16-bit PCM data into 8-bit data for transmission and storage purposes.

Discussion and Sample Code

The AudioUlawEncodeDecode02 program

This program is designed to illustrate 8-bit ULAW encoding and decoding, and to compare that process with simple truncation from sixteen to eight bits.

> *(Because the documentation for Sun's Java Sound API refers to mu-law as ULAW, I will frequently follow that lead and use the term ULAW in the discussion that follows.)*

Verification of the program

I have verified this program against results produced by Sun's API, and results produced by other non-Java programs available on the Internet. The 8-bit ULAW values produced by the program match the values produced by Sun's ULAW encoding algorithm. The 16-bit values produced by decoding the ULAW values also match the values produced by Sun's ULAW decoding algorithm. The decoding results also match the values produced by two different decoding algorithm implementations that I found on the web and used to verify this program.

Two sets of output data

The program produces two sets of output data. The first set shows the numeric results of simply truncating a series of sample values from sixteen bits to eight bits, preserving the eight most significant bits, and then restoring those truncated values back into a 16-bit format.

The second set of output data shows the numeric results of encoding the same set of 16-bit sample values into 8-bit ULAW values, and then decoding those ULAW values back into 16-bit values.

I will discuss the experimental results first, and then discuss the program used to produce those results.

The simple truncation results

The results of the truncation experiment are shown in Figure 1. Results were computed over a wide range of 16-bit linear PCM sample values, between zero and 24,577.

Each line of data in Figure 1 shows the original sample value, the 16-bit representation of the truncated value *(eight data bits in the eight most significant bits of a 16-bit value* *of type* ***short*)**, the difference between the original sample value and the truncated value, and that difference *(error)* expressed as a percent of the original sample value.

> *(Because of the difficulties encountered when dividing by zero, a percent error was not computed for a sample value of zero.)*

```
Process and display truncation
0 0 0
1 0 1 100.0%
2 0 2 100.0%
3 0 3 100.0%
4 0 4 100.0%
5 0 5 100.0%
7 0 7 100.0%
9 0 9 100.0%
13 0 13 100.0%
17 0 17 100.0%
25 0 25 100.0%
33 0 33 100.0%
49 0 49 100.0%
65 0 65 100.0%
97 0 97 100.0%
129 0 129 100.0%
193 0 193 100.0%
257 256 1 0.38910505%
385 256 129 33.506493%
513 512 1 0.19493178%
769 768 1 0.130039%
1025 1024 1 0.09756097%
1537 1536 1 0.06506181%
2049 2048 1 0.048804294%
3073 3072 1 0.03254149%
4097 4096 1 0.024408104%
6145 6144 1 0.016273392%
8193 8192 1 0.012205541%
12289 12288 1 0.008137358%
16385 16384 1 0.006103143%
24577 24576 1 0.004068845%
Figure 1
```

Framing the experiment

The allowable values of the original 16-bit samples range from negative 32,768 to positive 32,767 *(approximately 96 db at six db per bit).*

When truncating data by eliminating some of the data bits, it is always necessary to decide which bits can be thrown away with minimum damage to the data. For the case where meaningful data is contained in the largest values that can be written, the choice is usually to throw away bits at the least significant end. That is what was done here. The data was converted from sixteen bits to eight bits by discarding the eight least significant bits. The remaining eight bits were retained in the most significant eight bits of the 16-bit sample.

Analysis of results

For those who are familiar with this sort of thing, it will come as no surprise that the error was one-hundred percent for all values less than 256. In other words, all values less than 256 were discarded when the eight least significant bits were discarded. For the most part, the errors for values larger than 256 were relatively small.

Resulting useful dynamic range

As a result, the dynamic range of the data was reduced from approximately 96 db to 48 db *(again estimated at six db per bit).* The range of sound intensities that can be represented in the data was reduced from approximately one part in 32,768 to one part in 128. Whether or not this is acceptable depends entirely on your application.

Familiar to many readers

These results will be yesterday's news to many of you. I present them here simply to serve as a baseline for similar experimental results that I will present for ULAW encoding and compression in the following sections.

The code used to produce these results is so straightforward that I'm not going to bore you by discussing it here. You can view that code in Listing 13 near the end of the lesson.

The mu-law encoding and compression results

The results of the ULAW experiment are shown in Figure 2. Results were computed over a same wide range of 16-bit linear PCM sample values as in the truncation experiment.

Note that this data includes an additional column of data that is not included in the truncation data shown in Figure 1. In particular, this data also includes the ULAW byte value in hex notation.

Each line of data in Figure 2 shows

- The original sample value *(which is the same as for the truncation experiment)*
- The encoded ULAW byte value in hex notation
- The 16-bit value produced by decoding the ULAW byte
- The difference between the original sample value and the decoded value
- That difference *(error)* expressed as a percent of the original sample value

  (Again, because of the difficulties encountered when dividing by zero, a percent error was not computed for a sample value of zero.)

```
Process and display ULAW
```

```
0 0xff 0 0
1 0xff 0 1 100.0%
2 0xff 0 2 100.0%
3 0xff 0 3 100.0%
4 0xfe 8 -4 -100.0%
5 0xfe 8 -3 -60.0%
7 0xfe 8 -1 -14.285714%
9 0xfe 8 1 11.111111%
13 0xfd 16 -3 -23.076923%
17 0xfd 16 1 5.882353%
25 0xfc 24 1 4.0%
33 0xfb 32 1 3.030303%
49 0xf9 48 1 2.0408163%
65 0xf7 64 1 1.5384616%
97 0xf3 96 1 1.0309278%
129 0xef 132 -3 -2.3255813%
193 0xeb 196 -3 -1.5544041%
257 0xe7 260 -3 -1.1673151%
385 0xdf 396 -11 -2.857143%
513 0xdb 524 -11 -2.1442494%
769 0xd3 780 -11 -1.4304291%
1025 0xcd 1052 -27 -2.6341465%
1537 0xc5 1564 -27 -1.7566688%
2049 0xbe 2108 -59 -2.8794534%
3073 0xb6 3132 -59 -1.919948%
4097 0xaf 4092 5 0.12204052%
6145 0xa7 6140 5 0.08136696%
8193 0x9f 8316 -123 -1.5012816%
12289 0x97 12412 -123 -1.0008951%
16385 0x8f 16764 -379 -2.3130913%
24577 0x87 24956 -379 -1.5420922%
Figure 2
```

Analysis of results

In this case, the percent error, which begins at one-hundred percent for a sample value of 1, decreases very rapidly as the sample values increase.

Sample values of 1 through 5 are rendered essentially useless by the mu-law encode/decode process. I will explain a little later why this is true.

> *(Contrast this with simple truncation where the lowest 256 values were rendered useless.)*

Values above 17 are very useful

For all sample values above 17, the resulting values are very useful with the percent error being less than six percent.

Note that even though the percent error never gets as low as in the high-valued cases in Figure 1, the errors that do exist are spread more uniformly across a wider dynamic range than is the case in Figure 1. That is one of the main attractions of the ULAW encoding scheme. It provides tolerable errors across a wide dynamic range as contrasted to very low errors across a narrow dynamic range for the truncation process.

What is the useful dynamic range?

If we choose the value 7 as the low end of the useful dynamic range in Figure 2, this results in a dynamic range of approximately 78 db for ULAW encoding as compared to 48 db for simple truncation. Even if we push the low end up to the value 17 with less than six-percent error, this still results in a useful dynamic range of approximately 72 db for ULAW encoding.

Having discussed why, now discuss how

The Java Sound API provides classes that make it possible for you to easily create and use ULAW encoded streams. *(I will show you how to do that in a subsequent lesson.)*

The real scoop on OOP

One of the great things about OOP is that if you have confidence in the author of an available class, and you are satisfied that an object of the class will satisfy your needs, there is no need for you to reinvent the wheel and to rewrite code that has already been written, tested, and released.

In the following sections, I will show you how to write code to perform ULAW encoding and decoding. My purpose is not to encourage you to reinvent the wheel. By all means, use Sun's classes for this purpose if they will meet your needs.

My purpose in showing you this code is to help you to better understand the algorithms that are used for ULAW encoding and decoding. It is my belief that you will better understand the subtle aspects of those algorithms when you see them implemented in code than would be the case if I were to present them in some other format.

Now for the program

The program named **AudioUlawEncodeDecode02** begins in Listing 1. As usual, I will discuss this program in fragments. You can view a complete listing of the program in Listing 13 near the end of the lesson.

```
public class AudioUlawEncodeDecode02{
  static int value = 0;
  static int increment = 1;
  static int limit = 4;
  static short shortValue =
(short)value;

  public static void main(
                        String
args[]){
    System.out.println(
               "Process and display
truncation");
    processAndDisplayTruncation();

    System.out.println();
    System.out.println(
                     "Process and
display ULAW");
    //Reinitialize values in the
processing loop.
    value = 0;
    increment = 1;
    limit = 4;
    shortValue = (short)value;
    processAndDisplayUlaw();

  }//end main

Listing 1
```

The main method

Figure 1 shows the **main** method, whose purpose is to control the generation and display of the output data shown in Figures 1 and 2.

The most significant code in Listing 1 is the pair of calls to the methods named **processAndDisplayTruncation** and **processAndDisplayUlaw**. These two methods execute the loops, invoke the **encode** and **decode** methods, and format the output data for Figures 1 and 2.

The processAndDisplayUlaw method

Listing 2 shows an abbreviated listing for the **processAndDisplayUlaw** method. This method is very straightforward, so I have deleted much of the code in Listing 2 for brevity. *(You can view the entire method in Listing 13 near the end of the lesson.)*

```
  static void processAndDisplayUlaw(){
    while((shortValue >= 0) &
                           (shortValue
< 32000)){

      byte ulawByte =
encode(shortValue);
      short result = decode(ulawByte);


      //print code deleted for brevity
      //loop control code deleted for
brevity

    }//end while loop
  }//end processAndDisplayTruncation
```

Listing 2

Three parts

The method consists of three main parts:

- A part to encode and decode the data
- A part to print the results
- A part to control the loop and to create the **short** values to be encoded and decoded

While somewhat tedious, the last two parts are totally straightforward. Therefore, I have deleted those parts from Listing 2.

The encode and decode method calls

The most important part in Listing 2 is the pair of statements that invoke the methods named **encode** and **decode** to:

- Encode a 16-bit **short** sample value into an 8-bit ULAW encoded byte
- Decode the 8-bit ULAW encoded byte into a 16-bit **short** sample value.

The code in Listing 2 produces the output shown previously in Figure 2.

Very similar code is used to control and display the results of the truncation operation shown in Figure 1, so I won't discuss it here.

The encoding algorithm

The encoding algorithm is relatively straightforward, consisting of the following steps:

- Convert the 16-bit **short** sample from two's complement format to sign-magnitude format, saving the value of the algebraic sign in the process.
- Clip the magnitude to a value of 32635 to prevent integer arithmetic overflow when the BIAS value is added.
- Add a BIAS value of 132 to the magnitude. This guarantees that a 1-bit will appear somewhere in the exponent region of the 15-bit magnitude.
- The exponent region is the set of eight bits appearing immediately to the right of the sign bit. The next step is to find the position of the leftmost 1-bit in the exponent region, and to record the position of that bit by counting bits from right to left in the exponent region. This position value can range from 0 to 7, with a

position of 0 indicating that the bit appears in the rightmost bit position in the exponent region.

- Extract and save a four-bit mantissa consisting of the four bits immediately to the right of the leftmost 1-bit in the exponent region identified above.
- Construct an 8-bit byte in which the leftmost bit is the sign bit, the four rightmost bits are the mantissa, and the three bits in between contain the position of the exponent bit.
- Get the one's complement of the 8-bit byte, which becomes the ULAW encoded byte that represents the 16-bit sample.

An illustration of the bit patterns in general

Figure 3 illustrates the bit patterns described above. The leftmost character in each row in Figure 3 is the row number. The characters to the right of each row number depict binary bits.

The first two rows of characters each depict sixteen data bits with the sign bit on the left and the least significant data bit on the right. The third row of characters depicts eight data bits.

```
1   S E E E E E E E E - - - - - - -
2   S 0 0 0 0 0 1 M M M M - - - - -
3   S E E E M M M M

Figure 3
```

The exponent region

The first row in Figure 3 shows the 8-bit exponent region immediately to the right of the sign bit. The exponent region is indicated by the upper-case E characters in Figure 3. The dash characters in Figure 3 indicate that the bits have no particular designation in general.

The mantissa region

The second row of characters in Figure 3 shows the 4-bit mantissa region, assuming that the most significant non-zero data bit is in the position indicated by the 1. The mantissa region is indicated by the upper-case M characters in Figure 3.

The mu-law encoded byte before complementing

The third row of characters in Figure 3 shows the 8-bit byte constructed as a result of mu-law encoding. The sign bit is shown on the left. The 4-bit mantissa is shown on the right. The three bits in the middle contain a value that specifies the position of the leftmost non-zero bit in the exponent region of the original data sample.

(Note that the encoding process must actually deliver the one's complement of the 8-bit byte as the last step in the encoding process. The decoding process reverses the one's complement operation as the first step in the decoding process.)

Bit patterns for a specific case

Figure 4 shows a more specific representation of the general information shown in Figure 3. The first two rows in Figure 4 show the general bit designations for the exponent region and the mantissa region with the leftmost non-zero data bit being in the position of the 1-bit in the second row.

```
1   S E E E E E E E E - - - - - - -
2   S 0 0 0 0 0 1 M M M M - - - - -
3   0 0 0 0 0 0 1 0 0 1 1 0 0 0 0 1
4   0 0 0 0 0 0 1 0 1 1 1 0 0 1 0 1
5   S E E E M M M M
6   0 0 1 0 0 1 1 1
7   1 1 0 1 1 0 0 0

Figure 4
```

The decimal value 609

The third row in Figure 4 shows the decimal value 609 in 16-bit binary sign-magnitude notation.

(Note that for consistency, the leftmost non-zero data bit is in the same position in the second and third rows.)

After adding the bias value of 132

The fourth row shows the result of adding the bias value of 132, resulting in a binary-encoded vale of 741.

General form of a mu-law byte

The fifth row shows the general form of the 8-bit mu-law byte *(before performing a one's complement operation on the byte).*

The specific mu-law byte before complementing

The sixth row shows the result of encoding the data from the fourth row into a mu-law byte.

The complemented mu-law byte

However, the one's complement of this byte must be returned from the encoding process. The byte shown in the seventh row is the actual byte that would be returned from the encoding process after taking the one's complement. This value would be represented as **0xd8** in hexadecimal notation.

The decoded value

Although we haven't discussed the decoding algorithm yet, when this byte is decoded, it will return a 16-bit short value of 620 for an encoding/decoding error of about 1.8 percent.

What does all this mean?

The encoding process converts the 16-bit integer sample data to a kind of 8-bit floating-point data, consisting of a sign bit, a 3-bit exponent, and a 4-bit mantissa. That is why the encoding process maintains good dynamic range with tolerable error levels throughout the dynamic range.

The value of the data in the 16-bit format is represented by the 4-bit mantissa along with the exponent bit. The position of the exponent bit represents the most significant bit in the data contained in the 16-bit integer format. The mantissa represents the value of the four data bits immediately to the right of the most significant bit. In effect, the five most significant data bits from the 16-bit data are extracted and saved in the ULAW byte.

Represent the five most significant data bits

The important point is that these five bits represent the five most significant data bits *(exclusive of leading zeros)* for both large values and small values of the sample. The encoding process doesn't favor large values over small values, as is the case with simple truncation from 16 bits to 8 bits.

Three least significant bits are ignored

If you draw a picture similar to Figure 4 for a data value less than eight, you will see that the two least significant bits in the 16-bit sample data are totally ignored, and the third bit from the right is partially ignored by this encoding process.

> *(The bias bit that is added into the third bit from the right may cause a carry bit to propagate into the mantissa region for values of four or greater, so the third bit from the right isn't totally ignored.)*

The fact that these three bits are either totally or partially ignored explains why the values from 1 through 7 in Figure 2 had high percent error values. As a practical matter, the encoding process discards the two least significant bits, and partially throws away the third least significant bit.

The method named encode

The source code for the **encode** method begins in Listing 3. The implementation of this method is loosely based on discussions and code at **link1** and **link2** identified earlier.

```
  static byte encode(short sample){
    final short BIAS = 132;//0x84
    final short CLIP = 32635;//32767-
BIAS
```

Listing 3

As you should expect by now, this method receives an incoming parameter of type **short** and returns a value of type **byte**.

The code in Listing 3 defines two constants. As you will see later, the **BIAS** value is added to the sample after it is converted to sign-magnitude format. I haven't found an explanation for the need to add this bias, and the only explanation that I can give is that it guarantees that a 1-bit will occur somewhere in the exponent region.

The need for the **CLIP** is fairly obvious. Without it, the adding of the **BIAS** value could cause integer overflow. Thus, as you will see later, prior to adding the **BIAS**, the sample magnitude is clipped to the value shown in Listing 3.

Convert sample to sign-magnitude format

The code in Listing 4 converts the **short** sample to sign-magnitude format. It also gets and saves the algebraic sign in the process.

```
    int sign = sample & 0x8000;
    if(sign != 0){
      sample = (short)-sample;
      sign = 0x80;
    }//end if
```

Listing 4

The code in Listing 4 is straightforward, and shouldn't require further discussion.

Clip the sample and add the BIAS

The code in Listing 5 clips the sample value to a maximum value of 32,635 as discussed earlier.

```
    if(sample > CLIP) sample = CLIP;
    sample += BIAS;

Listing 5
```

Then the code in Listing 5 adds the **BIAS** value of 132 to the sample. As mentioned earlier, adding the **BIAS** guarantees that a 1-bit will occur in the exponent region of the data, even for small sample values.

The sample is now ready for conversion to an 8-bit byte.

Get the exponent value

The exponent value is the position of the first 1-bit to the right of the sign bit in the exponent region of the data. The code in Listing 6 finds the position of the first 1-bit to the right of the sign bit, counting from right to left in the exponent region.

```
    int exp;
    //Shift sign bit off to the left
    short temp = (short)(sample << 1);
    for(exp = 7; exp > 0; exp--){
      if((temp & 0x8000) != 0)
break;//found it
      temp = (short)(temp <<
1);//shift and loop
    }//end for loop

Listing 6
```

The exponent position *(value)* can range from 0 to 7. I could have used a table lookup to get the value *(as is the case in **link1**)* but I elected to use a **for** loop and to compute the value on the fly instead. *(That way, I didn't have to explain a bunch of mystery values in a table.)*

The code in Listing 6 is straightforward and shouldn't require a discussion.

Get the mantissa

The mantissa is the set of four bits immediately to the right of the leftmost 1-bit in the exponent region. The code in Listing 7 shifts those four bits to the four least significant bits of the 16-bit value.

```
    temp = (short)(sample >> (exp +
3));
    //Mask and save those four bits
    int mantis = temp & 0x000f;
```

Listing 7

Then the code in Listing 7 masks those four bits, causing the remaining 12 bits to be converted to zero.

The one's complement is required

For reasons that I am unable to explain on any logical basis, the **ULAW** byte delivered by the encoder must be the one's complement of the bit structure that I described earlier.

> *(The first thing that is done in a **ULAW** decoder is to complement it, thus reversing the process. I haven't read anything that explains why this is done.)*

Construct the ULAW byte

The code in Listing 8 constructs and complements the **ULAW** byte by performing the following steps:

- Set the sign bit in the most significant bit of the 8-bit byte. *(The current value of the variable named **sign** is either 0x00 or 0x80, with the latter representing a negative sign.)*
- Position the exponent value in the three bits to the immediate right of the sign bit. *(Recall that this value ranges from 0 to 7 and specifies the position of the most significant data bit in the sample data relative to the least significant end of the exponent region.)*
- Set the 4-bit mantissa in the four least significant bits of the byte.
- Complement and return the byte constructed according to the above steps.

```
    byte ulawByte = (byte)(sign | (exp
<< 4) |

mantis);
    return (byte)~ulawByte;
  }//end encode
```

Listing 8

That completes the discussion of the method named **encode**.

The method named decode

The method named **decode** begins in Listing 9. This Java implementation is based loosely on the code and discussion at **link1** and **link2**. Note however, that this implementation is significantly different from the material at either of those web sites.

```
  static short decode(byte ulawByte){

    ulawByte = (byte)(~ulawByte);

Listing 9
```

This method receives an incoming **ULAW** byte and returns a **short** value that is an estimate of the sample value that was used to produce the **ULAW** byte.

The code in Listing 9 begins by performing a one's complement operation on the incoming byte to reverse the complement operation performed as the final step in the encoding of that byte.

Get the sign, exponent, and mantissa

The code in Listing 10 performs the following operations:

- Get the sign bit by masking off the most significant bit of the byte and saving it in a variable named **sign**.
- Get the exponent value by masking off the three exponent bits and sifting them right to the least significant bit position in the variable named **exp**.
- Get the mantissa by masking the four least significant bits of the byte and saving them in the variable named **mantis**.

```
    int sign = ulawByte & 0x80;
    int exp = (ulawByte & 0x70) >> 4;
    int mantis = ulawByte & 0xf;

Listing 10
```

Construct the output

The code in Listing 11 constructs the 16-bit output value based on the values of the sign, the exponent, and the mantissa. This value is constructed as type **int** for convenience and then cast to type **short** when it is returned.

```
    int rawValue =
               (mantis << (12 - 8 +
(exp - 1))) +
                              (132 <<
exp) - 132;

Listing 11
```

An exercise for the student

This is where I am going to allow you to do some work on your own to help you develop an understanding of the behavior of the expression in Listing 11 *(otherwise known as leaving it as an exercise for the student).*

I suggest that you choose the first three values from one of the lines in Figure 2, such as the following values for example.

193 0xeb 196

In this case, the value to the left is the original 16-bit sample value. The hexadecimal value in the middle is the value of the **ULAW** byte produced by the encoder. The value on the right is the value produced by decoding the value in the middle. In other words, the value on the right is the 16-bit estimate of the value on the left, produced by first encoding and then decoding the value on the left.

Extract the sign, the exponent, and the mantissa from the value in the middle and combine them according to the expression in Listing 11. If you do it correctly, you should end up with a 16-bit value that matches the value on the right.

Change the sign if necessary and return

The code in Listing 12 converts the raw 16-bit value into a signed two's complement **short** value and returns that value.

```
    return (short)((sign != 0)
                          ? - rawValue :
rawValue);
  }//end decode
```

Listing 12

That ends the discussion of the decode method.

Run the Program

At this point, you may find it useful to run the program and perform some experiments on your own. For example, you may find it interesting to change the value of the **BIAS** value in the **encode** method to see how that effects the difference between the actual sample value and the estimate of the sample value produced by first encoding the sample value and then decoding the **ULAW** byte.

You should be able to copy the code from Listing 13 into your editor, compile it, and run it.

Summary

In this lesson, I taught you how to write a Java program to encode 16-bit linear PCM samples into 8-bit mu-law bytes. I also taught you how to write a Java program to decode 8-bit mu-law bytes back into 16-bit PCM samples.

I performed an experiment to determine the distribution of the resulting information loss across a dynamic range of approximately 96 decibels. I compared that distribution with the information loss produced by simply truncating 16-bit PCM samples into 8-bit data for transmission and storage purposes.

The truncation approach reduces the dynamic range of the 16-bit data to approximately 48 db, and provides very low information loss across that rage.

The mu-law approach maintains around 78 db of usable dynamic range, and provides tolerable information loss across that range.

The primary advantage of mu-law encoding over simple truncation is that mu-law encoding maintains significantly greater dynamic range while keeping information loss within tolerable limits.

What's Next?

In the next lesson, I will teach you how to use the Java Sound API to write **ULAW** and ALAW encoded files.

Complete Program Listing

A complete listing of the program is contained in Listing 13.

```
/*File AudioUlawEncodeDecode02.java
Copyright 2003, R.G.Baldwin

This program is designed to illustrate 8-bit ULAW
encoding and decoding, and to compare that
process with simple truncation from sixteen to
eight bits.

The 8-bit ULAW values produced by the program
match the values produced by Sun's ULAW encoding
algorithm.  The 16-bit values produced by
decoding the ULAW values match the values
produced by two different decoding algorithm
implementations that I found on the web and
used for verifying this program.

The program produces two sets of output data. The
first set shows the numeric results of simply
```

truncating a series of sample values from sixteen bits to eight bits, preserving the eight most significant bits, and then restoring those truncated values back into a 16-bit format.

The second set of output data shows the numeric results of encoding the same set of sample values into 8-bit ULAW values, and then decoding those ULAW values back into 16-bit values.

The results of the truncation experiment are shown in the following table. Each line in this output shows the original sample value, the 16-bit representation of the truncated value,the difference between the original sample value and the truncated value, and that difference (error) expressed as a percent of the original sample value. (Because of the difficulties encountered when dividing by zero, a percent error was not computed for a sample value of zero.)

```
Process and display truncation
0 0 0
1 0 1 100.0%
2 0 2 100.0%
3 0 3 100.0%
4 0 4 100.0%
5 0 5 100.0%
7 0 7 100.0%
9 0 9 100.0%
13 0 13 100.0%
17 0 17 100.0%
25 0 25 100.0%
33 0 33 100.0%
49 0 49 100.0%
65 0 65 100.0%
97 0 97 100.0%
129 0 129 100.0%
193 0 193 100.0%
257 256 1 0.38910505%
385 256 129 33.506493%
513 512 1 0.19493178%
769 768 1 0.130039%
1025 1024 1 0.09756097%
1537 1536 1 0.06506181%
2049 2048 1 0.048804294%
3073 3072 1 0.03254149%
4097 4096 1 0.024408104%
6145 6144 1 0.016273392%
8193 8192 1 0.012205541%
12289 12288 1 0.008137358%
16385 16384 1 0.006103143%
24577 24576 1 0.004068845%
```

The results of the ULAW experiment are shown in

```
the following table.  Each line in this output
shows the original sample value, the ULAW byte
value in hex notation, the 16-bit value produced
by decoding the ULAW byte, the difference between
the original sample value and the decoded value,
and that difference (error) expressed as a
percent of the original sample value.  (Because
of the difficulties encountered when dividing by
zero, a percent error was not computed for a
sample value of zero.)

Process and display ULAW
0 0xff 0 0
1 0xff 0 1 100.0%
2 0xff 0 2 100.0%
3 0xff 0 3 100.0%
4 0xfe 8 -4 -100.0%
5 0xfe 8 -3 -60.0%
7 0xfe 8 -1 -14.285714%
9 0xfe 8 1 11.111111%
13 0xfd 16 -3 -23.076923%
17 0xfd 16 1 5.882353%
25 0xfc 24 1 4.0%
33 0xfb 32 1 3.030303%
49 0xf9 48 1 2.0408163%
65 0xf7 64 1 1.5384616%
97 0xf3 96 1 1.0309278%
129 0xef 132 -3 -2.3255813%
193 0xeb 196 -3 -1.5544041%
257 0xe7 260 -3 -1.1673151%
385 0xdf 396 -11 -2.857143%
513 0xdb 524 -11 -2.1442494%
769 0xd3 780 -11 -1.4304291%
1025 0xcd 1052 -27 -2.6341465%
1537 0xc5 1564 -27 -1.7566688%
2049 0xbe 2108 -59 -2.8794534%
3073 0xb6 3132 -59 -1.919948%
4097 0xaf 4092 5 0.12204052%
6145 0xa7 6140 5 0.08136696%
8193 0x9f 8316 -123 -1.5012816%
12289 0x97 12412 -123 -1.0008951%
16385 0x8f 16764 -379 -2.3130913%
24577 0x87 24956 -379 -1.5420922%

Tested using SDK 1.4.1 under Win2000
************************************************/

public class AudioUlawEncodeDecode02{
  static int value = 0;
  static int increment = 1;
  static int limit = 4;
  static short shortValue = (short)value;

  public static void main(
                        String args[]){
```

```
    System.out.println(
                 "Process and display truncation");
    processAndDisplayTruncation();

    System.out.println();
    System.out.println(
                       "Process and display ULAW");
    //Reinitialize values in the processing loop.
    value = 0;
    increment = 1;
    limit = 4;
    shortValue = (short)value;
    processAndDisplayUlaw();

  }//end main
  //-------------------------------------------//

  static short truncate(short sample){
    //Mask 8 lsb
    return (short)(sample & 0xff00);
  }//end truncate
  //-------------------------------------------//

  //This encoding method is loosely based on
  // online material at:
  // http://www.speech.cs.cmu.edu/comp.speech/
  // Section2/Q2.7.html
  static byte encode(short sample){
    final short BIAS = 132;//0x84
    final short CLIP = 32635;//32767-BIAS

    //Convert sample to sign-magnitude
    int sign = sample & 0x8000;
    if(sign != 0){
      sample = (short)-sample;
      sign = 0x80;
    }//end if

    //Because of the bias that is added, allowing
    // a value larger than CLIP would result in
    // integer overflow, so clip it.
    if(sample > CLIP) sample = CLIP;

    //Convert from 16-bit linear PCM to ulaw
    //Adding this bias guarantees a 1 bit in the
    // exponent region of the data, which is the
    // eight bits to the right of the sign bit.
    sample += BIAS;

    //Exponent value is the position of the first
    // 1 to the right of the sign bit in the
    // exponent region of the data.
    //Find the position of the first 1 to the
    // right of the sign bit, counting from right
    // to left in the exponent region.  The
    // exponent position (value) can range from 0
    // to 7.
    //Could use a table lookup but will compute
    // on the fly instead because that is better
    // for teaching the algorithm.
    int exp;
    //Shift sign bit off to the left
    short temp = (short)(sample << 1);
    for(exp = 7; exp > 0; exp--){
      if((temp & 0x8000) != 0) break;//found it
      temp = (short)(temp << 1);//shift and loop
    }//end for loop

    //The mantissa is the four bits to the right
    // of the first 1 bit in the exponent region.
    // Shift those four bits to the four lsb of
    // the 16-bit value.
    temp = (short)(sample >> (exp + 3));
    //Mask and save those four bits
    int mantis = temp & 0x000f;
    //Construct the complement of the ulaw byte.
    //Set the sign bit in the msb of the 8-bit
    // byte.  The value of sign is either 0x00 or
    // 0x80.
    //Position the exponent in the three bits to
    // the right of the sign bit.
    //Set the 4-bit mantissa in the four lsb of
    // the byte.
    //Note that the one's complement of this
    // value will be returned.
    byte ulawByte = (byte)(sign | (exp << 4) |
                                          mantis);
    //Now complement to create actual ulaw byte
    // and return it.
    return (byte)~ulawByte;
  }//end encode
  //------------------------------------------//

  //This decode method is loosely based on
  // material at:
  // http://web.umr.edu/~dcallier/school/
  // 311_final_report.doc
  //That material was published by David Callier
  // and Chess Combites as a semester project and
  // has been reformulated into Java code by this
  // author..
  static short decode(byte ulawByte){
    //Perform one's complement to undo the one's
    // complement at the end of the encode
    // algorithm.
    ulawByte = (byte)(~ulawByte);
    //Get the sign bit from the ulawByte
    int sign = ulawByte & 0x80;
    //Get the value of the exponent in the three
    // bytes to the right of the sign bit.
    int exp = (ulawByte & 0x70) >> 4;
    //Get the mantissa by masking off and saving
    // the four lsb in the ulawByte.
    int mantis = ulawByte & 0xf;
    //Construct the 16-bit output value as type
    // int for simplicity and cast to short
    // before returning.
    int rawValue =
               (mantis << (12 - 8 + (exp - 1))) +
                              (132 << exp) - 132;
    //Change the sign if necessary and return
    // the 16-bit estimate of the original
    // sample value.
    return (short)((sign != 0)
                        ? - rawValue : rawValue);
  }//end decode
  //-------------------------------------------//

  static void processAndDisplayTruncation(){
    while((shortValue >= 0) &
                         (shortValue < 32000)){
      short result = truncate(shortValue);
      System.out.print(shortValue + " ");
      System.out.print(result + " ");
      System.out.print(shortValue - result);
      if(shortValue > 0){
        System.out.println(" " +
            ((float)(100.0*(shortValue - result)/
                             shortValue)) + "%");
      }else{
        System.out.println();
      }//end else
      value = value + increment;
      shortValue = (short)value;
      if(value > limit){
        increment *= 2;
        limit *= 2;
      }//end if
      if(increment > 32000)break;
    }//end while loop
  }//end processAndDisplayTruncation
  //-------------------------------------------//

  static void processAndDisplayUlaw(){
    while((shortValue >= 0) &
                         (shortValue < 32000)){
      byte ulawByte = encode(shortValue);
      short result = decode(ulawByte);
      System.out.print(shortValue + " ");
      System.out.print("0x" +
                       Integer.toHexString(
                       ulawByte & 0xff) + " ");
      System.out.print(result + " ");
      System.out.print(shortValue - result);
      if(shortValue > 0){
        System.out.println(" " +
            ((float)(100.0*(shortValue - result)/
                             shortValue)) + "%");
      }else{
        System.out.println();
      }//end else
      value = value + increment;
      shortValue = (short)value;
      if(value > limit){
        increment *= 2;
        limit *= 2;
      }//end if
      if(increment > 32000)break;
    }//end while loop
  }//end processAndDisplayTruncation
  //-------------------------------------------//
}//end class AudioUlawEncodeDecode02.java
```

Listing 13

About the author

Richard Baldwin is a college professor (at Austin Community College in Austin, TX) and private consultant whose primary focus is a combination of Java, C#, and XML. In addition to the many platform and/or language independent benefits of Java and C# applications, he believes that a combination of Java, C#, and XML will become the primary driving force in the delivery of structured information on the Web.

Richard has participated in numerous consulting projects and he frequently provides onsite training at the high-tech companies located in and around Austin, Texas. He is the author of Baldwin's Programming Tutorials, which has gained a worldwide following among experienced and aspiring programmers. He has also published articles in JavaPro magazine.

Richard holds an MSEE degree from Southern Methodist University and has many years of experience in the application of computer technology to real-world problems.

Baldwin@DickBaldwin.com

-end-