25. Practical: An ID3 Parser

With a library for parsing binary data, you're ready to write some code for reading and writing an actual binary format, that of ID3 tags. ID3 tags are used to embed metadata in MP3 audio files. Dealing with ID3 tags will be a good test of the binary data library because the ID3 format is a true real-world format--a mix of engineering trade-offs and idiosyncratic design choices that does, whatever else might be said about it, get the job done. In case you missed the file-sharing revolution, here's a quick overview of what ID3 tags are and how they relate to MP3 files.

MP3, also known as MPEG Audio Layer 3, is a format for storing compressed audio data, designed by researchers at Fraunhofer IIS and standardized by the Moving Picture Experts Group, a joint committee of the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC). However, the MP3 format, by itself, defines only how to store audio data. That's fine as long as all your MP3 files are managed by a single application that can store metadata externally and keep track of which metadata goes with which files. However, when people started passing around individual MP3 files on the Internet, via file-sharing systems such as Napster, they soon discovered they needed a way to embed metadata in the MP3 files themselves.

Because the MP3 standard was already codified and a fair bit of software and hardware had already been written that knew how to decode the existing MP3 format, any scheme for embedding information in an MP3 file would have to be invisible to MP3 decoders. Enter ID3.

The original ID3 format, invented by programmer Eric Kemp, consisted of 128 bytes stuck on the end of an MP3 file where it'd be ignored by most MP3 software. It consisted of four 30-character fields, one each for the song title, the album title, the artist name, and a comment; a four-byte year field; and a one-byte genre code. Kemp provided standard meanings for the first 80 genre codes. Nullsoft, the makers of Winamp, a popular MP3 player, later supplemented this list with another 60 or so genres.

This format was easy to parse but obviously quite limited. It had no way to encode names longer than 30 characters; it was limited to 256 genres, and the meaning of the genre codes had to be agreed upon by all users of ID3-aware software. There wasn't even a way to encode the CD track number of a particular MP3 file until another programmer, Michael Mutschler, proposed embedding the track number in the comment field, separated from the rest of the comment by a null byte, so existing ID3 software, which tended to read up to the first null in each of the text fields, would ignore it. Kemp's version is now called ID3v1, and Mutschler's is ID3v1.1.

Limited as they were, the version 1 proposals were at least a partial solution to the metadata problem, so they were adopted by many MP3 ripping programs (which had to put the ID3 tag into the MP3 files) and MP3 players (which would extract the information in the ID3 tag to display to the user).[1]

By 1998, however, the limitations were really becoming annoying, and a new group, led by Martin Nilsson, started work on a completely new tagging scheme, which came to be called ID3v2. The ID3v2 format is extremely flexible, allowing for many kinds of information to be included, with almost no length limitations. It also takes advantage of certain details of the MP3 format to allow ID3v2 tags to be placed at the beginning of an MP3 file.

ID3v2 tags are, however, more of a challenge to parse than version 1 tags. In this chapter, you'll use the binary data parsing library from the previous chapter to develop code that can read and write ID3v2 tags. Or at least you'll make a reasonable start--where ID3v1 was too simple, ID3v2 is baroque to the point of being completely overengineered. Implementing every nook and cranny of the specification, especially if you want to support all three versions that have been specified, would be a fair bit of work. However, you can ignore many of the features in those specifications since they're rarely used "in the wild." For starters, you can ignore, for now, a whole version, 2.4, since it has not been widely adopted and mostly just adds more needless flexibility compared to version 2.3. I'll focus on versions 2.2 and 2.3 because they're both widely used and are different enough from each other to keep things interesting.

Structure of an ID3v2 Tag

Before you can start cutting code, you'll need to be familiar with the overall structure of an ID3v2 tag. A tag starts with a header containing information about the tag as a whole. The first three bytes of the header encode the string "ID3" in ISO-8859-1 characters. In other words, they're the bytes 73, 68, and 51. Then comes two bytes that encode the major version and revision of the ID3 specification to which the tag purports to conform. They're followed by a single byte whose individual bits are treated as flags. The meanings of the individual flags depend on the version of the spec. Some of the flags can affect the way the rest of the tag is parsed. The "major version" is actually used to record the minor version of the spec, while the "revision" is the subminor version of the spec. Thus, the "major version" field for a tag conforming to the 2.3.0 spec is 3. The revision field is always zero since each new ID3v2 spec has bumped the minor version, leaving the subminor version at zero. The value stored in the major version field of the tag has, as you'll see, a dramatic effect on how you'll parse the rest of the tag.

The last field in the tag header is an integer, encoded in four bytes but using only seven bits from each byte, that gives the total size of the tag, not counting the header. In version 2.3 tags, the header may be followed by several *extended header* fields; otherwise, the remainder of the tag data is divided into *frames*. Different types of frames store different kinds of information, from simple textual information, such as the song name, to embedded images. Each frame starts with a header containing a string identifier and a size. In version 2.3, the frame header also contains two bytes worth of flags and, depending on the value of one the flags, an optional one-byte code indicating how the rest of the frame is encrypted.

Frames are a perfect example of a tagged data structure--to know how to parse the body of a frame, you need to read the header and use the identifier to determine what kind of frame you're reading.

The ID3 tag header contains no direct indication of how many frames are in a tag--the tag header tells you how big the tag is, but since many frames are variable length, the only way to find out how many frames the tag contains is to read the frame data. Also, the size given in the tag header may be larger than the actual number of bytes of frame data; the frames may be followed with enough null bytes to pad the tag out to the specified size. This makes it possible for tag editors to modify a tag without having to rewrite the whole MP3 file.[2]

So, the main issues you have to deal with are reading the ID3 header; determining whether you're reading a version 2.2 or 2.3 tag; and reading the frame data, stopping either when you've read the complete tag or when you've hit the padding bytes.

Defining a Package

Like the other libraries you've developed so far, the code you'll write in this chapter is worth putting in its own package. You'll need to refer to functions from both the binary data and pathname libraries developed in Chapters 24 and 15 and will also want to export the names of the functions that make up the public API to this package. The following package definition does all that:

```
(defpackage :com.gigamonkeys.id3v2
  (:use :common-lisp
        :com.gigamonkeys.binary-data
        :com.gigamonkeys.pathnames)
  (:export
   :read-id3
   :mp3-p
   :id3-p
   :album
   :composer
   :genre
   :encoding-program
   :artist
   :part-of-set
   :track
   :song
   :year
   :size
   :translated-genre))
```

As usual, you can, and probably should, change the `com.gigamonkeys` part of the package name to your own domain.

Integer Types

You can start by defining binary types for reading and writing several of the primitive types used by the ID3 format, various sizes of unsigned integers, and four kinds of strings.

ID3 uses unsigned integers encoded in one, two, three, and four bytes. If you first write a general `unsigned-integer` binary type that takes the number of bytes to read as an argument, you can then use the short form of `define-binary-type` to define the specific types. The general `unsigned-integer` type looks like this:

```
(define-binary-type unsigned-integer (bytes)
  (:reader (in)
    (loop with value = 0
       for low-bit downfrom (* 8 (1- bytes)) to 0 by 8 do
         (setf (ldb (byte 8 low-bit) value) (read-byte in))
       finally (return value)))
  (:writer (out value)
    (loop for low-bit downfrom (* 8 (1- bytes)) to 0 by 8
       do (write-byte (ldb (byte 8 low-bit) value) out))))
```

Now you can use the short form of `define-binary-type` to define one type for each size of integer used in the ID3 format like this:

```
(define-binary-type u1 () (unsigned-integer :bytes 1))
(define-binary-type u2 () (unsigned-integer :bytes 2))
(define-binary-type u3 () (unsigned-integer :bytes 3))
(define-binary-type u4 () (unsigned-integer :bytes 4))
```

Another type you'll need to be able to read and write is the 28-bit value used in the header. This size is encoded using 28 bits rather than a multiple of 8, such as 32 bits, because an ID3 tag can't contain the byte `#xff` followed by a byte with the top 3 bits on because that pattern has a special meaning to MP3 decoders. None of the other fields in the ID3 header could possibly contain such a byte sequence, but if you encoded the tag size as a regular `unsigned-integer`, it might. To avoid that possibility, the size is encoded using only the bottom seven bits of each byte, with the top bit always zero.[3]

Thus, it can be read and written a lot like an `unsigned-integer` except the size of the byte specifier you pass to **`LDB`** should be seven rather than eight. This similarity suggests that if you add a parameter, `bits-per-byte`, to the existing `unsigned-integer` binary type, you could then define a new type, `id3-tag-size`, using a short-form `define-binary-type`. The new version of `unsigned-integer` is just like the old version except with `bits-per-byte` used everywhere the old version hardwired the number eight. It looks like this:

```
(define-binary-type unsigned-integer (bytes bits-per-byte)
  (:reader (in)
    (loop with value = 0
       for low-bit downfrom (* bits-per-byte (1- bytes)) to 0 by bits-per-byte do
         (setf (ldb (byte bits-per-byte low-bit) value) (read-byte in))
       finally (return value)))
  (:writer (out value)
    (loop for low-bit downfrom (* bits-per-byte (1- bytes)) to 0 by bits-per-byte
       do (write-byte (ldb (byte bits-per-byte low-bit) value) out))))
```

The definition of `id3-tag-size` is then trivial.

```
(define-binary-type id3-tag-size () (unsigned-integer :bytes 4 :bits-per-byte 7))
```

You'll also have to change the definitions of `u1` through `u4` to specify eight bits per byte like this:

```
(define-binary-type u1 () (unsigned-integer :bytes 1 :bits-per-byte 8))
(define-binary-type u2 () (unsigned-integer :bytes 2 :bits-per-byte 8))
(define-binary-type u3 () (unsigned-integer :bytes 3 :bits-per-byte 8))
(define-binary-type u4 () (unsigned-integer :bytes 4 :bits-per-byte 8))
```

String Types

The other kinds of primitive types that are ubiquitous in the ID3 format are strings. In the previous chapter I discussed some of the issues you have to consider when dealing with strings in binary files, such as the difference between character codes and character encodings.

ID3 uses two different character codes, ISO 8859-1 and Unicode. ISO 8859-1, also known as Latin-1, is an eight-bit character code that extends ASCII with characters used by the languages of Western Europe. In other words, the code points from 0-127 map to the same characters in ASCII and ISO 8859-1, but ISO 8859-1 also provides mappings for code points up to 255. Unicode is a character code designed to provide a code point for virtually every character of all the world's languages. Unicode is a superset of ISO 8859-1 in the same way that ISO 8859-1 is a superset of ASCII--the code points from 0-255 map to the same characters in both ISO 8859-1 and Unicode. (Thus, Unicode is also a superset of ASCII.)

Since ISO 8859-1 is an eight-bit character code, it's encoded using one byte per character. For Unicode strings, ID3 uses the UCS-2 encoding with a leading *byte order mark*.[4] I'll discuss what a byte order mark is in a moment.

Reading and writing these two encodings isn't a problem--it's just a question of reading and writing unsigned integers in various formats, and you just finished writing the code to do that.

The trick is how you translate those numeric values to Lisp character objects.

The Lisp implementation you're using probably uses either Unicode or ISO 8859-1 as its internal character code. And since all the values from 0-255 map to the same characters in both ISO 8859-1 and Unicode, you can use Lisp's **`CODE-CHAR`** and **`CHAR-CODE`** functions to translate those values in both character codes. However, if your Lisp supports only ISO 8859-1, then you'll be able to represent only the first 255 Unicode characters as Lisp characters. In other words, in such a Lisp implementation, if you try to process an ID3 tag that uses Unicode strings and if any of those strings contain characters with code points higher than 255, you'll get an error when you try to translate the code point to a Lisp character. For now I'll assume either you're using a Unicode-based Lisp or you won't process any files containing characters outside the ISO 8859-1 range.

The other issue with encoding strings is how to know how many bytes to interpret as character data. ID3 uses two strategies I mentioned in the previous chapter--some strings are terminated with a null character, while other strings occur in positions where you can determine the number of bytes to read, either because the string at that position is always the same length or because the string is at the end of a composite structure whose overall size you know. Note, however, that the number of bytes isn't necessarily the same as the number of characters in the string.

Putting all these variations together, the ID3 format uses four ways to read and write strings--two characters crossed with two ways of delimiting the string data.

Obviously, much of the logic of reading and writing strings will be quite similar. So, you can start by defining two binary types, one for reading strings of a specific length (in characters) and another for reading terminated strings. Both types take advantage of that the type argument to `read-value` and `write-value` is just another piece of data; you can make the type of character to read a parameter of these types. This is a technique you'll use quite a few times in this chapter.

```
(define-binary-type generic-string (length character-type)
  (:reader (in)
    (let ((string (make-string length)))
      (dotimes (i length)
        (setf (char string i) (read-value character-type in)))
      string))
  (:writer (out string)
    (dotimes (i length)
      (write-value character-type out (char string i)))))

(define-binary-type generic-terminated-string (terminator character-type)
  (:reader (in)
    (with-output-to-string (s)
      (loop for char = (read-value character-type in)
            until (char= char terminator) do (write-char char s))))
  (:writer (out string)
    (loop for char across string
          do (write-value character-type out char)
          finally (write-value character-type out terminator))))
```

With these types available, there's not much to reading ISO 8859-1 strings. Because the `character-type` argument you pass to `read-value` and `write-value` of a `generic-string` must be the name of a binary type, you need to define an `iso-8859-1-char` binary type. This also gives you a good place to put a bit of sanity checking on the code points of characters you read and write.

```
(define-binary-type iso-8859-1-char ()
  (:reader (in)
    (let ((code (read-byte in)))
      (or (code-char code)
          (error "Character code ~d not supported" code))))
  (:writer (out char)
    (let ((code (char-code char)))
      (if (<= 0 code #xff)
          (write-byte code out)
          (error "Illegal character for iso-8859-1 encoding: character: ~c with code: ~d" char
```

Now defining the ISO 8859-1 string types is trivial using the short form of `define-binary-type` as follows:

```
(define-binary-type iso-8859-1-string (length)
  (generic-string :length length :character-type 'iso-8859-1-char))

(define-binary-type iso-8859-1-terminated-string (terminator)
  (generic-terminated-string :terminator terminator :character-type 'iso-8859-1-char))
```

Reading UCS-2 strings is only slightly more complex. The complexity arises because you can encode a UCS-2 code point in two ways: most significant byte first (big-endian) or least significant byte first (little-endian). UCS-2 strings therefore start with two extra bytes, called the *byte order mark*, made up of the numeric value `#xfeff` encoded in either big-endian form or little-endian form. When reading a UCS-2 string, you read the byte order mark and then, depending on its value, read either big-endian or little-endian characters. Thus, you'll need two different UCS-2 character types. But you need only one version of the sanity-checking code, so you can define a parameterized binary type like this:

```
(define-binary-type ucs-2-char (swap)
  (:reader (in)
    (let ((code (read-value 'u2 in)))
      (when swap (setf code (swap-bytes code)))
      (or (code-char code) (error "Character code ~d not supported" code))))
  (:writer (out char)
    (let ((code (char-code char)))
      (unless (<= 0 code #xffff)
        (error "Illegal character for ucs-2 encoding: ~c with char-code: ~d" char code))
      (when swap (setf code (swap-bytes code)))
      (write-value 'u2 out code))))
```

where the `swap-bytes` function can be defined as follows, taking advantage of **`LDB`** being **`SETF`**able and thus **`ROTATEF`**able:

```
(defun swap-bytes (code)
  (assert (<= code #xffff))
  (rotatef (ldb (byte 8 0) code) (ldb (byte 8 8) code))
  code)
```

Using `ucs-2-char`, you can define two character types that will be used as the `character-type` arguments to the generic string functions.

```
(define-binary-type ucs-2-char-big-endian () (ucs-2-char :swap nil))

(define-binary-type ucs-2-char-little-endian () (ucs-2-char :swap t))
```

Then you need a function that returns the name of the character type to use based on the value of the byte order mark.

```
(defun ucs-2-char-type (byte-order-mark)
  (ecase byte-order-mark
    (#xfeff 'ucs-2-char-big-endian)
    (#xfffe 'ucs-2-char-little-endian)))
```

Now you can define length- and terminator-delimited string types for UCS-2-encoded strings that read the byte order mark and use it to determine which variant of UCS-2 character to pass as the `character-type` argument to `read-value` and `write-value`. The only other wrinkle is that you need to translate the `length` argument, which is a number of bytes, to the number of characters to read, accounting for the byte order mark.

```
(define-binary-type ucs-2-string (length)
  (:reader (in)
    (let ((byte-order-mark (read-value 'u2 in))
          (characters (1- (/ length 2))))
      (read-value
       'generic-string in
       :length characters
       :character-type (ucs-2-char-type byte-order-mark))))
  (:writer (out string)
    (write-value 'u2 out #xfeff)
    (write-value
     'generic-string out string
     :length (length string)
     :character-type (ucs-2-char-type #xfeff))))

(define-binary-type ucs-2-terminated-string (terminator)
  (:reader (in)
    (let ((byte-order-mark (read-value 'u2 in)))
      (read-value
       'generic-terminated-string in
       :terminator terminator
       :character-type (ucs-2-char-type byte-order-mark))))
  (:writer (out string)
    (write-value 'u2 out #xfeff)
    (write-value
     'generic-terminated-string out string
     :terminator terminator
     :character-type (ucs-2-char-type #xfeff))))
```

ID3 Tag Header

With the basic primitive types done, you're ready to switch to a high-level view and start defining binary classes to represent first the ID3 tag as a whole and then the individual frames.

If you turn first to the ID3v2.2 specification, you'll see that the basic structure of the tag is this header:

```
ID3/file identifier      "ID3"
ID3 version              $02 00
ID3 flags                %xx000000
ID3 size             4 * %0xxxxxxx
```

followed by frame data and padding. Since you've already defined binary types to read and write all the fields in the header, defining a class that can read the header of an ID3 tag is just a matter of putting them together.

```
(define-binary-class id3-tag ()
  ((identifier     (iso-8859-1-string :length 3))
   (major-version  u1)
   (revision       u1)
   (flags          u1)
   (size           id3-tag-size)))
```

If you have some MP3 files lying around, you can test this much of the code and also see what version of ID3 tags your MP3s contain. First you can write a function that reads an `id3-tag`, as just defined, from the beginning of a file. Be aware, however, that ID3 tags aren't required to appear at the beginning of a file, though these days they almost always do. To find an ID3 tag elsewhere in a file, you can scan the file looking for the sequence of bytes 73, 68, 51 (in other words, the string "ID3").[5] For now you can probably get away with assuming the tags are the first thing in the file.

```
(defun read-id3 (file)
  (with-open-file (in file :element-type '(unsigned-byte 8))
    (read-value 'id3-tag in)))
```

On top of this function you can build a function that takes a filename and prints the information in the tag header along with the name of the file.

```
(defun show-tag-header (file)
  (with-slots (identifier major-version revision flags size) (read-id3 file)
    (format t "~a ~d.~d ~8,'0b ~d bytes -- ~a~%"
            identifier major-version revision flags size (enough-namestring file))))
```

It prints output that looks like this:

```
ID3V2> (show-tag-header "/usr2/mp3/Kitka/Wintersongs/02 Byla Cesta.mp3")
ID3 2.0 00000000 2165 bytes -- Kitka/Wintersongs/02 Byla Cesta.mp3
NIL
```

Of course, to determine what versions of ID3 are most common in your MP3 library, it'd be handier to have a function that returns a summary of all the MP3 files under a given directory. You can write one easily enough using the `walk-directory` function defined in Chapter 15. First define a helper function that tests whether a given filename has an `mp3` extension.

```
(defun mp3-p (file)
  (and
   (not (directory-pathname-p file))
   (string-equal "mp3" (pathname-type file))))
```

Then you can combine `show-tag-header` and `mp3-p` with `walk-directory` to print a summary of the ID3 header in each file under a given directory.

```
(defun show-tag-headers (dir)
  (walk-directory dir #'show-tag-header :test #'mp3-p))
```

However, if you have a lot of MP3s, you may just want a count of how many ID3 tags of each version you have in your MP3 collection. To get that information, you might write a function like this:

```
(defun count-versions (dir)
  (let ((versions (mapcar #'(lambda (x) (cons x 0)) '(2 3 4))))
    (flet ((count-version (file)
             (incf (cdr (assoc (major-version (read-id3 file)) versions)))))
      (walk-directory dir #'count-version :test #'mp3-p))
    versions))
```

Another function you'll need in Chapter 29 is one that tests whether a file actually starts with an ID3 tag, which you can define like this:

```
(defun id3-p (file)
  (with-open-file (in file :element-type '(unsigned-byte 8))
    (string= "ID3" (read-value 'iso-8859-1-string in :length 3))))
```

ID3 Frames

As I discussed earlier, the bulk of an ID3 tag is divided into frames. Each frame has a structure similar to that of the tag as a whole. Each frame starts with a header indicating what kind of frame it is and the size of the frame in bytes. The structure of the frame header changed slightly between version 2.2 and version 2.3 of the ID3 format, and eventually you'll have to deal with both forms. To start, you can focus on parsing version 2.2 frames.

The header of a 2.2 frame consists of three bytes that encode a three-character ISO 8859-1 string followed by a three-byte unsigned integer, which specifies the size of the frame in bytes, excluding the six-byte header. The string identifies what type of frame it is, which determines how you parse the data following the size. This is exactly the kind of situation for which you defined the `define-tagged-binary-class` macro. You can define a tagged class that reads the frame header and then dispatches to the appropriate concrete class using a function that maps IDs to a class names.

```
(define-tagged-binary-class id3-frame ()
  ((id (iso-8859-1-string :length 3))
   (size u3))
  (:dispatch (find-frame-class id)))
```

Now you're ready to start implementing concrete frame classes. However, the specification defines quite a few--63 in version 2.2 and even more in later specs. Even considering frame types that share a common structure to be equivalent, you'll still find 24 unique frame types in version 2.2. But only a few of these are used "in the wild." So rather than immediately setting to work defining classes for each of the frame types, you can start by writing a generic frame class that lets you read the frames in a tag without parsing the data within the frames themselves. This will give you a way to find out what frames are actually present in the MP3s you want to process. You'll need this class eventually anyway because the specification allows for experimental frames that you'll need to be able to read without parsing.

Since the size field of the frame header tells you exactly how many bytes long the frame is, you can define a `generic-frame` class that extends `id3-frame` and adds a single field, `data`, that will hold an array of bytes.

```
(define-binary-class generic-frame (id3-frame)
  ((data (raw-bytes :size size))))
```

The type of the data field, `raw-bytes`, just needs to hold an array of bytes. You can define it like this:

```
(define-binary-type raw-bytes (size)
  (:reader (in)
    (let ((buf (make-array size :element-type '(unsigned-byte 8))))
      (read-sequence buf in)
      buf))
  (:writer (out buf)
    (write-sequence buf out)))
```

For the time being, you'll want all frames to be read as `generic-frames`, so you can define the `find-frame-class` function used in `id3-frame`'s `:dispatch` expression to always return `generic-frame`, regardless of the frame's `id`.

```
(defun find-frame-class (id)
  (declare (ignore id))
  'generic-frame)
```

Now you need to modify `id3-tag` so it'll read frames after the header fields. There's only one tricky bit to reading the frame data: although the tag header tells you how many bytes long the tag is, that number includes the padding that can follow the frame data. Since the tag header doesn't tell you how many frames the tag contains, the only way to tell when you've hit the padding is to look for a null byte where you'd expect a frame identifier.

To handle this, you can define a binary type, `id3-frames`, that will be responsible for reading the remainder of a tag, creating frame objects to represent all the frames it finds, and then skipping over any padding. This type will take as a parameter the tag size, which it can use to avoid reading past the end of the tag. But the reading code will also need to detect the beginning of the padding that can follow the tag's frame data. Rather than calling `read-value` directly in `id3-frames :reader`, you should use a function `read-frame`, which you'll define to return **`NIL`** when it detects padding, otherwise returning an `id3-frame` object read using `read-value`. Assuming you define `read-frame` so it reads only one byte past the end of the last frame in order to detect the start of the padding, you can define the `id3-frames` binary type like this:

```
(define-binary-type id3-frames (tag-size)
  (:reader (in)
    (loop with to-read = tag-size
          while (plusp to-read)
          for frame = (read-frame in)
          while frame
```

```
           do (decf to-read (+ 6 (size frame)))
           collect frame
           finally (loop repeat (1- to-read) do (read-byte in))))
  (:writer (out frames)
    (loop with to-write = tag-size
           for frame in frames
           do (write-value 'id3-frame out frame)
           (decf to-write (+ 6 (size frame)))
           finally (loop repeat to-write do (write-byte 0 out)))))
```

You can use this type to add a `frames` slot to `id3-tag`.

```
(define-binary-class id3-tag ()
  ((identifier     (iso-8859-1-string :length 3))
   (major-version  u1)
   (revision       u1)
   (flags          u1)
   (size           id3-tag-size)
   (frames         (id3-frames :tag-size size))))
```

Detecting Tag Padding

Now all that remains is to implement `read-frame`. This is a bit tricky since the code that actually reads bytes from the stream is several layers down from `read-frame`.

What you'd really like to do in `read-frame` is read one byte and return **`NIL`** if it's a null and otherwise read a frame with `read-value`. Unfortunately, if you read the byte in `read-frame`, then it won't be available to be read by `read-value`.[6]

It turns out this is a perfect opportunity to use the condition system--you can check for null bytes in the low-level code that reads from the stream and signal a condition when you read a null; `read-frame` can then handle the condition by unwinding the stack before more bytes are read. In addition to turning out to be a tidy solution to the problem of detecting the start of the tag's padding, this is also an example of how you can use conditions for purposes other than handling errors.

You can start by defining a condition type to be signaled by the low-level code and handled by the high-level code. This condition doesn't need any slots--you just need a distinct class of condition so you know no other code will be signaling or handling it.

```
(define-condition in-padding () ())
```

Next you need to define a binary type whose `:reader` reads a given number of bytes, first reading a single byte and signaling an `in-padding` condition if the byte is null and otherwise reading the remaining bytes as an `iso-8859-1-string` and combining it with the first byte read.

```
(define-binary-type frame-id (length)
  (:reader (in)
    (let ((first-byte (read-byte in)))
      (when (= first-byte 0) (signal 'in-padding))
      (let ((rest (read-value 'iso-8859-1-string in :length (1- length))))
        (concatenate
         'string (string (code-char first-byte)) rest))))
  (:writer (out id)
    (write-value 'iso-8859-1-string out id :length length)))
```

If you redefine `id3-frame` to make the type of its `id` slot `frame-id` instead of `iso-8859-1-string`, the condition will be signaled whenever `id3-frame`'s `read-value` method reads a null byte instead of the beginning of a frame.

```
(define-tagged-binary-class id3-frame ()
  ((id (frame-id :length 3))
   (size u3))
  (:dispatch (find-frame-class id)))
```

Now all `read-frame` has to do is wrap a call to `read-value` in a **`HANDLER-CASE`** that handles the `in-padding` condition by returning **`NIL`**.

```
(defun read-frame (in)
  (handler-case (read-value 'id3-frame in)
    (in-padding () nil)))
```

With `read-frame` defined, you can now read a complete version 2.2 ID3 tag, representing frames with instances of `generic-frame`. In the "What Frames Do You Actually Need?" section, you'll do some experiments at the REPL to determine what frame classes you need to implement. But first let's add support for version 2.3 ID3 tags.

Supporting Multiple Versions of ID3

Currently, `id3-tag` is defined using `define-binary-class`, but if you want to support multiple versions of ID3, it makes more sense to use a `define-tagged-binary-class` that dispatches on the `major-version` value. As it turns out, all versions of ID3v2 have the same structure up to the size field. So, you can define a tagged binary class like the following that defines this basic structure and then dispatches to the appropriate version-specific subclass:

```
(define-tagged-binary-class id3-tag ()
  ((identifier     (iso-8859-1-string :length 3))
   (major-version  u1)
   (revision       u1)
   (flags          u1)
   (size           id3-tag-size))
  (:dispatch
   (ecase major-version
     (2 'id3v2.2-tag)
     (3 'id3v2.3-tag))))
```

Version 2.2 and version 2.3 tags differ in two ways. First, the header of a version 2.3 tag may be extended with up to four optional extended header fields, as determined by values in the flags field. Second, the frame format changed between version 2.2 and version 2.3, which means you'll have to use different classes to represent version 2.2 frames and the corresponding version 2.3 frames.

Since the new `id3-tag` class is based on the one you originally wrote to represent version 2.2 tags, it's not surprising that the new `id3v2.2-tag` class is trivial, inheriting most of its slots from the new `id3-tag` class and adding the one missing slot, `frames`. Because version 2.2 and version 2.3 tags use different frame formats, you'll have to change the `id3-frames` type to be parameterized with the type of frame to read. For now, assume you'll do that and add a `:frame-type` argument to the `id3-frames` type descriptor like this:

```
(define-binary-class id3v2.2-tag (id3-tag)
  ((frames (id3-frames :tag-size size :frame-type 'id3v2.2-frame))))
```

The `id3v2.3-tag` class is slightly more complex because of the optional fields. The first three of the four optional fields are included when the sixth bit in `flags` is set. They're a four-byte integer specifying the size of the extended header, two bytes worth of flags, and another four-byte integer specifying how many bytes of padding are included in the tag.[7] The fourth optional field, included when the fifteenth bit of the extended header flags is set, is a four-byte cyclic redundancy check (CRC) of the rest of the tag.

The binary data library doesn't provide any special support for optional fields in a binary class, but it turns out that regular parameterized binary types are sufficient. You can define a type parameterized with the name of a type and a value that indicates whether a value of that type should actually be read or written.

```
(define-binary-type optional (type if)
  (:reader (in)
    (when if (read-value type in)))
  (:writer (out value)
    (when if (write-value type out value))))
```

Using `if` as the parameter name looks a bit strange in that code, but it makes the `optional` type descriptors quite readable. For instance, here's the definition of `id3v2.3-tag` using `optional` slots:

```
(define-binary-class id3v2.3-tag (id3-tag)
  ((extended-header-size (optional :type 'u4 :if (extended-p flags)))
   (extra-flags          (optional :type 'u2 :if (extended-p flags)))
   (padding-size         (optional :type 'u4 :if (extended-p flags)))
   (crc                  (optional :type 'u4 :if (crc-p flags extra-flags)))
   (frames               (id3-frames :tag-size size :frame-type 'id3v2.3-frame))))
```

where `extended-p` and `crc-p` are helper functions that test the appropriate bit of the flags value they're passed. To test whether an individual bit of an integer is set, you can use **`LOGBITP`**, another bit-twiddling function. It takes an index and an integer and returns true if the specified bit is set in the integer.

```
(defun extended-p (flags) (logbitp 6 flags))

(defun crc-p (flags extra-flags)
  (and (extended-p flags) (logbitp 15 extra-flags)))
```

As in the version 2.2 tag class, the frames slot is defined to be of type `id3-frames`, passing the name of the frame type as a parameter. You do, however, need to make a few small changes to `id3-frames` and `read-frame` to support the extra `frame-type` parameter.

```
(define-binary-type id3-frames (tag-size frame-type)
  (:reader (in)
    (loop with to-read = tag-size
          while (plusp to-read)
          for frame = (read-frame frame-type in)
          while frame
          do (decf to-read (+ (frame-header-size frame) (size frame)))
          collect frame
          finally (loop repeat (1- to-read) do (read-byte in))))
  (:writer (out frames)
    (loop with to-write = tag-size
          for frame in frames
          do (write-value frame-type out frame)
          (decf to-write (+ (frame-header-size frame) (size frame)))
          finally (loop repeat to-write do (write-byte 0 out)))))

(defun read-frame (frame-type in)
  (handler-case (read-value frame-type in)
    (in-padding () nil)))
```

The changes are in the calls to `read-frame` and `write-value`, where you need to pass the `frame-type` argument and, in computing the size of the frame, where you need to use a function `frame-header-size` instead of the literal value `6` since the frame header changed size between version 2.2 and version 2.3. Since the difference in the result of this function is based on the class of the frame, it makes sense to define it as a generic function like this:

```
(defgeneric frame-header-size (frame))
```

You'll define the necessary methods on that generic function in the next section after you define the new frame classes.

Versioned Frame Base Classes

Where before you defined a single base class for all frames, you'll now have two classes, `id3v2.2-frame` and `id3v2.3-frame`. The `id3v2.2-frame` class will be essentially the same as the original `id3-frame` class.

```
(define-tagged-binary-class id3v2.2-frame ()
  ((id (frame-id :length 3))
   (size u3))
  (:dispatch (find-frame-class id)))
```

The `id3v2.3-frame`, on the other hand, requires more changes. The frame identifier and size fields were extended in version 2.3 from three to four bytes each, and two bytes worth of flags were added. Additionally, the frame, like the version 2.3 tag, can contain optional fields, controlled by the values of three of the frame's flags.[8] With those changes in mind, you can define the version 2.3 frame base class, along with some helper functions, like this:

```
(define-tagged-binary-class id3v2.3-frame ()
  ((id                (frame-id :length 4))
   (size              u4)
   (flags             u2)
   (decompressed-size (optional :type 'u4 :if (frame-compressed-p flags)))
   (encryption-scheme (optional :type 'u1 :if (frame-encrypted-p flags)))
   (grouping-identity (optional :type 'u1 :if (frame-grouped-p flags))))
  (:dispatch (find-frame-class id)))

(defun frame-compressed-p (flags) (logbitp 7 flags))

(defun frame-encrypted-p (flags) (logbitp 6 flags))

(defun frame-grouped-p (flags) (logbitp 5 flags))
```

With these two classes defined, you can now implement the methods on the generic function `frame-header-size`.

```
(defmethod frame-header-size ((frame id3v2.2-frame)) 6)

(defmethod frame-header-size ((frame id3v2.3-frame)) 10)
```

The optional fields in a version 2.3 frame aren't counted as part of the header for this computation since they're already included in the value of the frame's `size`.

Versioned Concrete Frame Classes

In the original definition, `generic-frame` subclassed `id3-frame`. But now `id3-frame` has been replaced with the two version-specific base classes, `id3v2.2-frame` and `id3v2.3-frame`. So, you need to define two new versions of `generic-frame`, one for each base class. One way to define this classes would be like this:

```
(define-binary-class generic-frame-v2.2 (id3v2.2-frame)
  ((data (raw-bytes :size size))))

(define-binary-class generic-frame-v2.3 (id3v2.3-frame)
  ((data (raw-bytes :size size))))
```

However, it's a bit annoying that these two classes are the same except for their superclass. It's not too bad in this case since there's only one additional field. But if you take this approach for other concrete frame classes, ones that have a more complex internal structure that's identical between the two ID3 versions, the duplication will be more irksome.

Another approach, and the one you should actually use, is to define a class `generic-frame` as a *mixin*: a class intended to be used as a superclass along with one of the version-specific base classes to produce a concrete, version-specific frame class. The only tricky bit about this approach is that if `generic-frame` doesn't extend either of the frame base classes, then you can't refer to the `size` slot in its definition. Instead, you must use the `current-binary-object` function I discussed at the end of the previous chapter to access the object you're in the midst of reading or writing and pass it to `size`. And you need to account for the difference in the number of bytes of the total frame size that will be left over, in the case of a version 2.3 frame, if any of the optional fields are included in the frame. So, you should define a generic function `data-bytes` with methods that do the right thing for both version 2.2 and version 2.3 frames.

```
(define-binary-class generic-frame ()
  ((data (raw-bytes :size (data-bytes (current-binary-object))))))

(defgeneric data-bytes (frame))

(defmethod data-bytes ((frame id3v2.2-frame))
  (size frame))

(defmethod data-bytes ((frame id3v2.3-frame))
  (let ((flags (flags frame)))
    (- (size frame)
       (if (frame-compressed-p flags) 4 0)
       (if (frame-encrypted-p flags) 1 0)
       (if (frame-grouped-p flags) 1 0))))
```

Then you can define concrete classes that extend one of the version-specific base classes and `generic-frame` to define version-specific generic frame classes.

```
(define-binary-class generic-frame-v2.2 (id3v2.2-frame generic-frame) ())

(define-binary-class generic-frame-v2.3 (id3v2.3-frame generic-frame) ())
```

With these classes defined, you can redefine the `find-frame-class` function to return the right versioned class based on the length of the identifier.

```
(defun find-frame-class (id)
  (ecase (length id)
    (3 'generic-frame-v2.2)
    (4 'generic-frame-v2.3)))
```

What Frames Do You Actually Need?

With the ability to read both version 2.2 and version 2.3 tags using generic frames, you're ready to start implementing classes to represent the specific frames you care about. However, before you dive in, you should take a breather and figure out what frames you actually care about since, as I mentioned earlier, the ID3 spec specifies many frames that are almost never used. Of course, what frames you care about depends on what kinds of applications you're interested in writing. If you're mostly interested in extracting information from existing ID3 tags, then you need implement only the classes representing the frames containing the information you care about. On the other hand, if you want to write an ID3 tag editor, you may need to support all the frames.

Rather than guessing which frames will be most useful, you can use the code you've already written to poke around a bit at the REPL and see what frames are actually used in your own MP3s. To start, you need an instance of `id3-tag`, which you can get with the `read-id3` function.

```
ID3V2> (read-id3 "/usr2/mp3/Kitka/Wintersongs/02 Byla Cesta.mp3")
#<ID3V2.2-TAG @ #x727b2912>
```

Since you'll want to play with this object a bit, you should save it in a variable.

```
ID3V2> (defparameter *id3* (read-id3 "/usr2/mp3/Kitka/Wintersongs/02 Byla Cesta.mp3"))
*ID3*
```

Now you can see, for example, how many frames it has.

```
ID3V2> (length (frames *id3*))
11
```

Not too many--let's take a look at what they are.

```
ID3V2> (frames *id3*)
(#<GENERIC-FRAME-V2.2 @ #x72dabdda> #<GENERIC-FRAME-V2.2 @ #x72dabec2>
 #<GENERIC-FRAME-V2.2 @ #x72dabfa2> #<GENERIC-FRAME-V2.2 @ #x72dac08a>
 #<GENERIC-FRAME-V2.2 @ #x72dac16a> #<GENERIC-FRAME-V2.2 @ #x72dac24a>
 #<GENERIC-FRAME-V2.2 @ #x72dac32a> #<GENERIC-FRAME-V2.2 @ #x72dac40a>
 #<GENERIC-FRAME-V2.2 @ #x72dac4f2> #<GENERIC-FRAME-V2.2 @ #x72dac632>
 #<GENERIC-FRAME-V2.2 @ #x72dac7b2>)
```

Okay, that's not too informative. What you really want to know are what kinds of frames are in there. In other words, you want to know the `id`s of those frames, which you can get with a simple **`MAPCAR`** like this:

```
ID3V2> (mapcar #'id (frames *id3*))
("TT2" "TP1" "TAL" "TRK" "TPA" "TYE" "TCO" "TEN" "COM" "COM" "COM")
```

If you look up these identifiers in the ID3v2.2 spec, you'll discover that all the frames with identifiers starting with *T* are text information frames and have a similar structure. And *COM* is the identifier for comment frames, which have a structure similar to that of text information frames. The particular text information frames identified here turn out to be the frames for representing the song title, artist, album, track, part of set, year, genre, and encoding program.

Of course, this is just one MP3 file. Maybe other frames are used in other files. It's easy enough to discover. First define a function that combines the previous **`MAPCAR`** expression with a call to `read-id3` and wraps the whole thing in a **`DELETE-DUPLICATES`** to keep things tidy. You'll have to use a `:test` argument of `#'string=` to **`DELETE-DUPLICATES`** to specify that you want two elements considered the same if they're the same string.

```
(defun frame-types (file)
  (delete-duplicates (mapcar #'id (frames (read-id3 file))) :test #'string=))
```

This should give the same answer except with only one of each identifier when passed the same filename.

```
ID3V2> (frame-types "/usr2/mp3/Kitka/Wintersongs/02 Byla Cesta.mp3")
("TT2" "TP1" "TAL" "TRK" "TPA" "TYE" "TCO" "TEN" "COM")
```

Then you can use Chapter 15's `walk-directory` function along with `mp3-p` to find every MP3 file under a directory and combine the results of calling `frame-types` on each file. Recall that **`NUNION`** is the recycling version of the **`UNION`** function; since `frame-types` makes a new list for each file, this is safe.

```
(defun frame-types-in-dir (dir)
  (let ((ids ()))
    (flet ((collect (file)
             (setf ids (nunion ids (frame-types file) :test #'string=))))
      (walk-directory dir #'collect :test #'mp3-p))
    ids))
```

Now pass it the name of a directory, and it'll tell you the set of identifiers used in all the MP3 files under that directory. It may take a few seconds depending how many MP3 files you have, but you'll probably get something similar to this:

```
ID3V2> (frame-types-in-dir "/usr2/mp3/")
("TCON" "COMM" "TRCK" "TIT2" "TPE1" "TALB" "TCP" "TT2" "TP1" "TCM"
 "TAL" "TRK" "TPA" "TYE" "TCO" "TEN" "COM")
```

The four-letter identifiers are the version 2.3 equivalents of the version 2.2 identifiers I discussed previously. Since the information stored in those frames is exactly the information you'll need in Chapter 27, it makes sense to implement classes only for the frames actually used, namely, text information and comment frames, which you'll do in the next two sections. If you decide later that you want to support other frame types, it's mostly a matter of translating the ID3 specifications into the appropriate binary class definitions.

Text Information Frames

All text information frames consist of two fields: a single byte indicating which string encoding is used in the frame and a string encoded in the remaining bytes of the frame. If the encoding byte is zero, the string is encoded in ISO 8859-1; if the encoding is one, the string is a UCS-2 string.

You've already defined binary types representing the four different kinds of strings--two different encodings each with two different methods of delimiting the string. However, `define-binary-class` provides no direct facility for determining the type of value to read based on other values in the object. Instead, you can define a binary type that you pass the value of the encoding byte and that then reads or writes the appropriate kind of string.

As long as you're defining such a type, you can also define it to take two parameters, `:length` and `:terminator`, and pick the right type of string based on which argument is supplied. To implement this new type, you must first define some helper functions. The first two return the name of the appropriate string type based on the encoding byte.

```
(defun non-terminated-type (encoding)
  (ecase encoding
    (0 'iso-8859-1-string)
    (1 'ucs-2-string)))

(defun terminated-type (encoding)
  (ecase encoding
    (0 'iso-8859-1-terminated-string)
    (1 'ucs-2-terminated-string)))
```

Then `string-args` uses the encoding byte, the length, and the terminator to determine several of the arguments to be passed to `read-value` and `write-value` by the `:reader` and `:writer` of `id3-encoded-string`. One of the length and terminator arguments to `string-args` should always be **`NIL`**.

```
(defun string-args (encoding length terminator)
  (cond
    (length
     (values (non-terminated-type encoding) :length length))
    (terminator
     (values (terminated-type encoding) :terminator terminator))))
```

With those helpers, the definition of `id3-encoded-string` is simple. One detail to note is that the keyword--either `:length` or `:terminator`--used in the call to `read-value` and `write-value` is just another piece of data returned by `string-args`. Although keywords in arguments lists are almost always literal keywords, they don't have to be.

```
(define-binary-type id3-encoded-string (encoding length terminator)
  (:reader (in)
    (multiple-value-bind (type keyword arg)
        (string-args encoding length terminator)
      (read-value type in keyword arg)))
  (:writer (out string)
    (multiple-value-bind (type keyword arg)
        (string-args encoding length terminator)
      (write-value type out string keyword arg))))
```

Now you can define a `text-info` mixin class, much the way you defined `generic-frame` earlier.

```
(define-binary-class text-info-frame ()
  ((encoding u1)
   (information (id3-encoded-string :encoding encoding :length (bytes-left 1)))))
```

As when you defined `generic-frame`, you need access to the size of the frame, in this case to compute the `:length` argument to pass to `id3-encoded-string`. Because you'll need to do a similar computation in the next class you define, you can go ahead and define a helper function, `bytes-left`, that uses `current-binary-object` to get at the size of the frame.

```
(defun bytes-left (bytes-read)
  (- (size (current-binary-object)) bytes-read))
```

Now, as you did with the `generic-frame` mixin, you can define two version-specific concrete classes with a minimum of duplicated code.

```
(define-binary-class text-info-frame-v2.2 (id3v2.2-frame text-info-frame) ())

(define-binary-class text-info-frame-v2.3 (id3v2.3-frame text-info-frame) ())
```

To wire these classes in, you need to modify `find-frame-class` to return the appropriate class name when the ID indicates the frame is a text information frame, namely, whenever the ID starts with *T* and isn't *TXX* or *TXXX*.

```
(defun find-frame-class (name)
  (cond
    ((and (char= (char name 0) #\T)
          (not (member name '("TXX" "TXXX") :test #'string=)))
     (ecase (length name)
       (3 'text-info-frame-v2.2)
       (4 'text-info-frame-v2.3)))
    (t
     (ecase (length name)
       (3 'generic-frame-v2.2)
       (4 'generic-frame-v2.3)))))
```

Comment Frames

Another commonly used frame type is the comment frame, which is like a text information frame with a few extra fields. Like a text information frame, it starts with a single byte indicating the string encoding used in the frame. That byte is followed by a three-character ISO 8859-1 string (regardless of the value of the string encoding byte), which indicates what language the comment is in using an ISO-639-2 code, for example, "eng" for English or "jpn" for Japanese. That field is followed by two strings encoded as indicated by the first byte. The first is a null-terminated string containing a description of the comment. The second, which takes up the remainder of the frame, is the comment text itself.

```
(define-binary-class comment-frame ()
  ((encoding u1)
   (language (iso-8859-1-string :length 3))
   (description (id3-encoded-string :encoding encoding :terminator +null+))
   (text (id3-encoded-string
          :encoding encoding
          :length (bytes-left
                   (+ 1 ; encoding
                      3 ; language
                      (encoded-string-length description encoding t)))))))
```

As in the definition of the `text-info` mixin, you can use `bytes-left` to compute the size of the final string. However, since the `description` field is a variable-length string, the number of bytes read prior to the start of `text` isn't a constant. To make matters worse, the number of bytes used to encode `description` is dependent on the encoding. So, you should define a helper function that returns the number of bytes used to encode a string given the string, the encoding code, and a boolean indicating whether the string is terminated with an extra character.

```
(defun encoded-string-length (string encoding terminated)
  (let ((characters (+ (length string) (if terminated 1 0))))
    (* characters (ecase encoding (0 1) (1 2)))))
```

And, as before, you can define the concrete version-specific comment frame classes and wire them into `find-frame-class`.

```
(define-binary-class comment-frame-v2.2 (id3v2.2-frame comment-frame) ())

(define-binary-class comment-frame-v2.3 (id3v2.3-frame comment-frame) ())

(defun find-frame-class (name)
  (cond
    ((and (char= (char name 0) #\T)
          (not (member name '("TXX" "TXXX") :test #'string=)))
     (ecase (length name)
       (3 'text-info-frame-v2.2)
       (4 'text-info-frame-v2.3)))
    ((string= name "COM")  'comment-frame-v2.2)
    ((string= name "COMM") 'comment-frame-v2.3)
    (t
     (ecase (length name)
       (3 'generic-frame-v2.2)
       (4 'generic-frame-v2.3)))))
```

Extracting Information from an ID3 Tag

Now that you have the basic ability to read and write ID3 tags, you have a lot of directions you could take this code. If you want to develop a complete ID3 tag editor, you'll need to implement specific classes for all the frame types. You'd also need to define methods for manipulating the tag and frame objects in a consistent way (for instance, if you change the value of a string in a `text-info-frame`, you'll likely need to adjust the size); as the code stands, there's nothing to make sure that happens.[9]

Or, if you just need to extract certain pieces of information about an MP3 file from its ID3 tag--as you will when you develop a streaming MP3 server in Chapters 27, 28, and 29--you'll need to write functions that find the appropriate frames and extract the information you want.

Finally, to make this production-quality code, you'd have to pore over the ID3 specs and deal with the details I skipped over in the interest of space. In particular, some of the flags in both the tag and the frame can affect the way the contents of the tag or frame is read; unless you write some code that does the right thing when those flags are set, there may be ID3 tags that this code won't be able to parse correctly. But the code from this chapter should be capable of parsing nearly all the MP3s you actually encounter.

For now you can finish with a few functions to extract individual pieces of information from an `id3-tag`. You'll need these functions in Chapter 27 and probably in other code that uses this library. They belong in this library because they depend on details of the ID3 format that the users of this library shouldn't have to worry about.

To get, say, the name of the song of the MP3 from which an `id3-tag` was extracted, you need to find the ID3 frame with a specific identifier and then extract the information field. And some pieces of information, such as the genre, can require further decoding. Luckily, all the frames that contain the information you'll care about are text information frames, so extracting a particular piece of information mostly boils down to using the right identifier to look up the appropriate frame. Of course, the ID3 authors decided to change all the identifiers between ID3v2.2 and ID3v2.3, so you'll have to account for that.

Nothing too complex--you just need to figure out the right path to get to the various pieces of information. This is a perfect bit of code to develop interactively, much the way you figured out what frame classes you needed to implement. To start, you need an `id3-tag` object to play with. Assuming you have an MP3 laying around, you can use `read-id3` like this:

```
ID3V2> (defparameter *id3* (read-id3 "Kitka/Wintersongs/02 Byla Cesta.mp3"))
*ID3*
ID3V2> *id3*
#<ID3V2.2-TAG @ #x73d04c1a>
```

replacing `Kitka/Wintersongs/02 Byla Cesta.mp3` with the filename of your MP3. Once you have your `id3-tag` object, you can start poking around. For instance, you can check out the list of frame objects with the `frames` function.

```
ID3V2> (frames *id3*)
(#<TEXT-INFO-FRAME-V2.2 @ #x73d04cca>
 #<TEXT-INFO-FRAME-V2.2 @ #x73d04dba>
 #<TEXT-INFO-FRAME-V2.2 @ #x73d04ea2>
 #<TEXT-INFO-FRAME-V2.2 @ #x73d04f9a>
 #<TEXT-INFO-FRAME-V2.2 @ #x73d05082>
 #<TEXT-INFO-FRAME-V2.2 @ #x73d0516a>
 #<TEXT-INFO-FRAME-V2.2 @ #x73d05252>
 #<TEXT-INFO-FRAME-V2.2 @ #x73d0533a>
 #<COMMENT-FRAME-V2.2 @ #x73d0543a>
 #<COMMENT-FRAME-V2.2 @ #x73d05612>
 #<COMMENT-FRAME-V2.2 @ #x73d0586a>)
```

Now suppose you want to extract the song title. It's probably in one of those frames, but to find it, you need to find the frame with the "TT2" identifier. Well, you can check easily enough to see if the tag contains such a frame by extracting all the identifiers like this:

```
ID3V2> (mapcar #'id (frames *id3*))
("TT2" "TP1" "TAL" "TRK" "TPA" "TYE" "TCO" "TEN" "COM" "COM" "COM")
```

There it is, the first frame. However, there's no guarantee it'll always be the first frame, so you should probably look it up by identifier rather than position. That's also straightforward using the **`FIND`** function.

```
ID3V2> (find "TT2" (frames *id3*) :test #'string= :key #'id)
#<TEXT-INFO-FRAME-V2.2 @ #x73d04cca>
```

Now, to get at the actual information in the frame, do this:

```
ID3V2> (information (find "TT2" (frames *id3*) :test #'string= :key #'id))
"Byla Cesta^@"
```

Whoops. That `^@` is how Emacs prints a null character. In a maneuver reminiscent of the kludge that turned ID3v1 into ID3v1.1, the `information` slot of a text information frame, though not officially a null-terminated string, can contain a null, and ID3 readers are supposed to ignore any characters after the null. So, you need a function that takes a string and returns the contents up to the first null character, if any. That's easy enough using the `+null+` constant from the binary data library.

```
(defun upto-null (string)
  (subseq string 0 (position +null+ string)))
```

Now you can get just the title.

```
ID3V2> (upto-null (information (find "TT2" (frames *id3*) :test #'string= :key #'id)))
"Byla Cesta"
```

You could just wrap that code in a function named `song` that takes an `id3-tag` as an argument, and you'd be done. However, the only difference between this code and the code you'll use to extract the other pieces of information you'll need (such as the album name, the artist, and the genre) is the identifier. So, it's better to split up the code a bit. For starters, you can write a function that just finds a frame given an `id3-tag` and an identifier like this:

```
(defun find-frame (id3 id)
  (find id (frames id3) :test #'string= :key #'id))

ID3V2> (find-frame *id3* "TT2")
#<TEXT-INFO-FRAME-V2.2 @ #x73d04cca>
```

Then the other bit of code, the part that extracts the information from a `text-info-frame`, can go in another function.

```
(defun get-text-info (id3 id)
  (let ((frame (find-frame id3 id)))
    (when frame (upto-null (information frame)))))

ID3V2> (get-text-info *id3* "TT2")
"Byla Cesta"
```

Now the definition of `song` is just a matter of passing the right identifier.

```
(defun song (id3) (get-text-info id3 "TT2"))

ID3V2> (song *id3*)
"Byla Cesta"
```

However, this definition of `song` works only with version 2.2 tags since the identifier changed from "TT2" to "TIT2" between version 2.2 and version 2.3. And all the other tags changed too. Since the user of this library shouldn't have to know about different versions of the ID3 format to do something as simple as get the song title, you should probably handle those details for them. A simple way is to change `find-frame` to take not just a single identifier but a list of identifiers like this:

```
(defun find-frame (id3 ids)
  (find-if #'(lambda (x) (find (id x) ids :test #'string=)) (frames id3)))
```

Then change `get-text-info` slightly so it can take one or more identifiers using a **`&rest`** parameter.

```
(defun get-text-info (id3 &rest ids)
  (let ((frame (find-frame id3 ids)))
    (when frame (upto-null (information frame)))))
```

Then the change needed to allow `song` to support both version 2.2 and version 2.3 tags is just a matter of adding the version 2.3 identifier.

```
(defun song (id3) (get-text-info id3 "TT2" "TIT2"))
```

Then you just need to look up the appropriate version 2.2 and version 2.3 frame identifiers for any fields for which you want to provide an accessor function. Here are the ones you'll need in Chapter 27:

```
(defun album (id3) (get-text-info id3 "TAL" "TALB"))

(defun artist (id3) (get-text-info id3 "TP1" "TPE1"))

(defun track (id3) (get-text-info id3 "TRK" "TRCK"))

(defun year (id3) (get-text-info id3 "TYE" "TYER" "TDRC"))

(defun genre (id3) (get-text-info id3 "TCO" "TCON"))
```

The last wrinkle is that the way the `genre` is stored in the TCO or TCON frames isn't always human readable. Recall that in ID3v1, genres were stored as a single byte that encoded a particular genre from a fixed list. Unfortunately, those codes live on in ID3v2--if the text of the genre frame is a number in parentheses, the number is supposed to be interpreted as an ID3v1 genre code. But, again, users of this library probably won't care about that ancient history. So, you should provide a function that automatically translates the genre. The following function uses the `genre` function just defined to extract the actual genre text and then checks whether it starts with a left parenthesis, decoding the version 1 genre code with a function you'll define in a moment if it does:

```
(defun translated-genre (id3)
  (let ((genre (genre id3)))
    (if (and genre (char= #\( (char genre 0)))
      (translate-v1-genre genre)
      genre)))
```

Since a version 1 genre code is effectively just an index into an array of standard names, the easiest way to implement `translate-v1-genre` is to extract the number from the genre string and use it as an index into an actual array.

```
(defun translate-v1-genre (genre)
  (aref *id3-v1-genres* (parse-integer genre :start 1 :junk-allowed t)))
```

Then all you need to do is to define the array of names. The following array of names includes the 80 official version 1 genres plus the genres created by the authors of Winamp:

```
(defparameter *id3-v1-genres*
  #(
    ;; These are the official ID3v1 genres.
    "Blues" "Classic Rock" "Country" "Dance" "Disco" "Funk" "Grunge"
    "Hip-Hop" "Jazz" "Metal" "New Age" "Oldies" "Other" "Pop" "R&B" "Rap"
    "Reggae" "Rock" "Techno" "Industrial" "Alternative" "Ska"
    "Death Metal" "Pranks" "Soundtrack" "Euro-Techno" "Ambient"
    "Trip-Hop" "Vocal" "Jazz+Funk" "Fusion" "Trance" "Classical"
    "Instrumental" "Acid" "House" "Game" "Sound Clip" "Gospel" "Noise"
    "AlternRock" "Bass" "Soul" "Punk" "Space" "Meditative"
    "Instrumental Pop" "Instrumental Rock" "Ethnic" "Gothic" "Darkwave"
    "Techno-Industrial" "Electronic" "Pop-Folk" "Eurodance" "Dream"
    "Southern Rock" "Comedy" "Cult" "Gangsta" "Top 40" "Christian Rap"
    "Pop/Funk" "Jungle" "Native American" "Cabaret" "New Wave"
    "Psychadelic" "Rave" "Showtunes" "Trailer" "Lo-Fi" "Tribal"
    "Acid Punk" "Acid Jazz" "Polka" "Retro" "Musical" "Rock & Roll"
    "Hard Rock"

    ;; These were made up by the authors of Winamp but backported into
    ;; the ID3 spec.
    "Folk" "Folk-Rock" "National Folk" "Swing" "Fast Fusion"
    "Bebob" "Latin" "Revival" "Celtic" "Bluegrass" "Avantgarde"
    "Gothic Rock" "Progressive Rock" "Psychedelic Rock" "Symphonic Rock"
    "Slow Rock" "Big Band" "Chorus" "Easy Listening" "Acoustic" "Humour"
    "Speech" "Chanson" "Opera" "Chamber Music" "Sonata" "Symphony"
    "Booty Bass" "Primus" "Porn Groove" "Satire" "Slow Jam" "Club"
    "Tango" "Samba" "Folklore" "Ballad" "Power Ballad" "Rhythmic Soul"
    "Freestyle" "Duet" "Punk Rock" "Drum Solo" "A capella" "Euro-House"
    "Dance Hall"

    ;; These were also invented by the Winamp folks but ignored by the
    ;; ID3 authors.
    "Goa" "Drum & Bass" "Club-House" "Hardcore" "Terror" "Indie"
    "BritPop" "Negerpunk" "Polsk Punk" "Beat" "Christian Gangsta Rap"
    "Heavy Metal" "Black Metal" "Crossover" "Contemporary Christian"
    "Christian Rock" "Merengue" "Salsa" "Thrash Metal" "Anime" "Jpop"
    "Synthpop"))
```

Once again, it probably feels like you wrote a ton of code in this chapter. But if you put it all in a file, or if you download the version from this book's Web site, you'll see it's just not that many lines--most of the pain of writing this library stems from having to understand the intricacies of the ID3 format itself. Anyway, now you have a major piece of what you'll turn into a streaming

MP3 server in Chapters 27, 28, and 29. The other major bit of infrastructure you'll need is a way to write server-side Web software, the topic of the next chapter.

---

[1]*Ripping* is the process by which a song on an audio CD is converted to an MP3 file on your hard drive. These days most ripping software also automatically retrieves information about the songs being ripped from online databases such as Gracenote (n�e the Compact Disc Database [CDDB]) or FreeDB, which it then embeds in the MP3 files as ID3 tags.

[2]Almost all file systems provide the ability to overwrite existing bytes of a file, but few, if any, provide a way to add or remove data at the beginning or middle of a file without having to rewrite the rest of the file. Since ID3 tags are typically stored at the beginning of a file, to rewrite an ID3 tag without disturbing the rest of the file you must replace the old tag with a new tag of exactly the same length. By writing ID3 tags with a certain amount of padding, you have a better chance of being able to do so--if the new tag has more data than the original tag, you use less padding, and if it's shorter, you use more.

[3]The frame data following the ID3 header could also potentially contain the illegal sequence. That's prevented using a different scheme that's turned on via one of the flags in the tag header. The code in this chapter doesn't account for the possibility that this flag might be set; in practice it's rarely used.

[4]In ID3v2.4, UCS-2 is replaced by the virtually identical UTF-16, and UTF-16BE and UTF-8 are added as additional encodings.

[5]The 2.4 version of the ID3 format also supports placing a footer at the end of a tag, which makes it easier to find a tag appended to the end of a file.

[6]Character streams support two functions, **`PEEK-CHAR`** and **`UNREAD-CHAR`**, either of which would be a perfect solution to this problem, but binary streams support no equivalent functions.

[7]If a tag had an extended header, you could use this value to determine where the frame data should end. However, if the extended header isn't used, you'd have to use the old algorithm anyway, so it's not worth adding code to do it another way.

[8]These flags, in addition to controlling whether the optional fields are included, can affect the parsing of the rest of the tag. In particular, if the seventh bit of the flags is set, then the actual frame data is compressed using the zlib algorithm, and if the sixth bit is set, the data is encrypted. In practice these options are rarely, if ever, used, so you can get away with ignoring them for now. But that would be an area you'd have to address to make this a production-quality ID3 library. One simple half solution would be to change `find-frame-class` to accept a second argument and pass it the flags; if the frame is compressed or encrypted, you could instantiate a generic frame to hold the data.

[9]Ensuring that kind of interfield consistency would be a fine application for `:after` methods on the accessor generic functions. For instance, you could define this `:after` method to keep `size` in sync with the `information` string:

```
(defmethod (setf information) :after (value (frame text-info-frame))
  (declare (ignore value))
  (with-slots (encoding size information) frame
    (setf size (encoded-string-length information encoding nil))))
```