Java Art Chapter 7. Video Watermarking with Barcodes

Watermarking aims to embed information, such as copyright details, in a video so it's too difficult to remove without adversely affecting the video's quality. It's a complicated task because there's always a tradeoff between a mark's robustness (its resistance to removal) and the modified video's fidelity (its visual quality). Adding a large copyright banner diagonally across every movie frame is certainly robust, but somewhat impacts the viewing experience.

A successful watermark has to survive many forms of attack, such as image rotation, cropping, resampling, time editing, compression, re-recording, noise addition, and color/contrast changes. One reason for making a watermark invisible (or nearly so) is to force an attacker to carry out drastic, hopefully costly, video-wide editing; when the watermark can't be located, it's hard to carry out a more focused attack . Inserting multiple copies of a watermark makes complete deletion harder as well, but repeating a mark too often may make it easier to detect.

Spatial watermarking modifies the pixels of a video frame, while frequency-domain marks affect the coefficients of a frame's frequency equations (e.g. as obtained with Discrete Fourier Transforms).

In this chapter, I add a hard-to-see (but not invisible) spatial watermark to video, repeated at intervals through the clip. The mark is disguised as a flickering line along the top of the movie frame, imitating the noise/interference that appears on poor video recordings. The mark is implemented as a series of very thin barcodes using the popular Code 128 format (http://en.wikipedia.org/wiki/Code_128). I chose barcodes since they offer the chance of a video mark being read optically as well as digitally.

Another aim of this chapter is to complain (a little) about the inadequacies of Sun's JMF (the Java Media Framework). Instead I employ the Xuggler API (http://www.xuggle.com/xuggler/) to edit and analyze video.

I develop two applications here: AddVideoMsg.java which adds barcodes representing a 'secret' message to a video, and GetVideoMsg.java which retrieves the message. Figure 1 illustrates how AddVideoMsg works, with an enlarged view of a modified frame with a barcode.

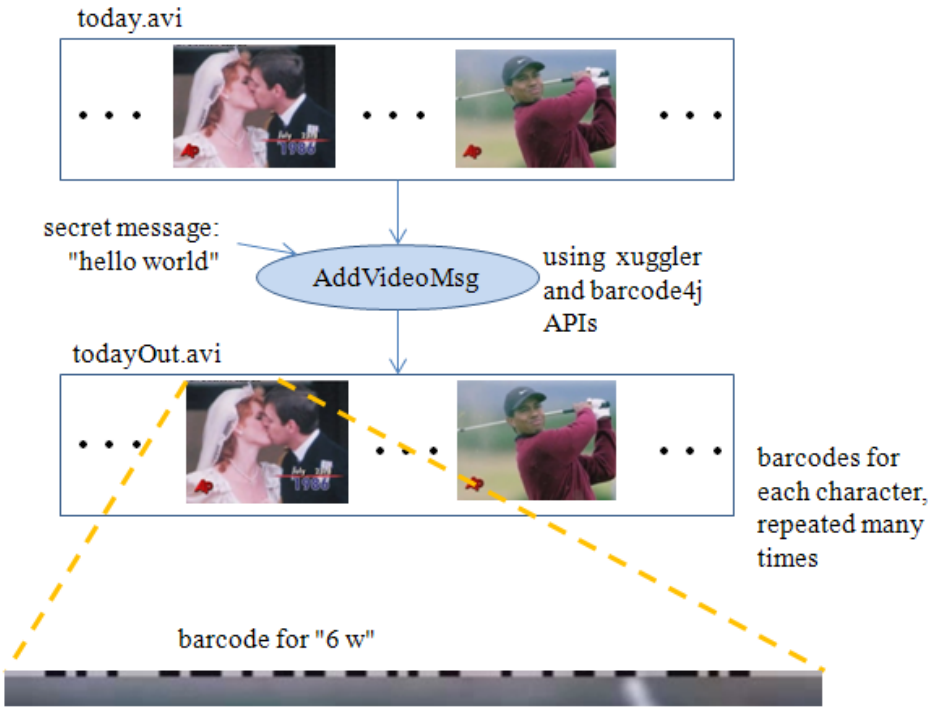

Figure 1. AddVideoMsg in Action.

The input video is "today.avi", and is modified to become "todayOut.avi". The video is manipulated using Xuggler, and the barcodes generated with the Barcode4J API (http://barcode4j.sourceforge.net/). The details of how the message is translated into barcodes and added to the video are explained below.

More on Watermarking

The Wikipedia page on digital watermarking (http://en.wikipedia.org/wiki/Digital_watermarking) summarizes the topic nicely and has a good set of links to more information. Another good summary is a technical overview on video watermarking at Microsoft (http://download.microsoft.com/download/d/6/b/d6bde980-5568-4926-8c71-dea63befed64/video_watermk.doc). *Digital Watermarking and Steganography*, 2nd ed., Ingemar Cox et al, Morgan Kaufmann, 2007 is a textbook on watermarking.

1. Editing Video in Java: JMF or Not?

Video editing doesn't have to be a complex programming task in Java, depending on which multimedia library you choose. Probably the best known, and certainly the oldest, is Sun Microsystems' JMF (the Java Media Framework; http://java.sun.com/javase/technologies/desktop/media/jmf/), which has been around since the late 1990's.

Unfortunately, JMF hasn't been substantially updated for a decade. But let's not be ageist: in some respects, the 'maturity' of the API doesn't matter, since it's

functionality has always been good, offering playback, capture, and processing of a range of audio and video formats. It was remarkable in it's day for supporting streaming content, pluggable codecs, transcoding, and platform-specific "performance packs" containing native-code optimizations.

The main problem caused by JMF's age is its lack of support for modern formats and codecs, such as MPEG-4, Windows Media, RealMedia, and most kinds of QuickTime and Flash content. This has led to the appearance of several alternative APIs, including FMJ (http://fmj-sf.net/), gstreamer (http://code.google.com/p/gstreamer-java/), fobs4JMF (http://fobs.sourceforge.net/), and Xuggler (http://www.xuggle.com/xuggler/). A list can be found on the Wikipedia page for JMF at http://en.wikipedia.org/wiki/Java_Media_Framework#Criticism_and_alternatives.

My watermarking needs are best matched by Xuggler: it offers easy ways to uncompress, modify, and re-compress a wide range of media formats, by utilizing FFmpeg behind the scenes (http://www.ffmpeg.org/). It comes with a well-designed high-level MediaTools API, that hides tedious aspects of encoding/decoding multiple formats and dealing with video processing states.

Xuggler's 'aloof' MediaTools API stands in stark contrast to the 'down-and-dirty' JMF, which requires the programmer to write plenty of boiler-plate for a common task like video frame editing. Figure 2 attempts to give a visual summary of the work involved.

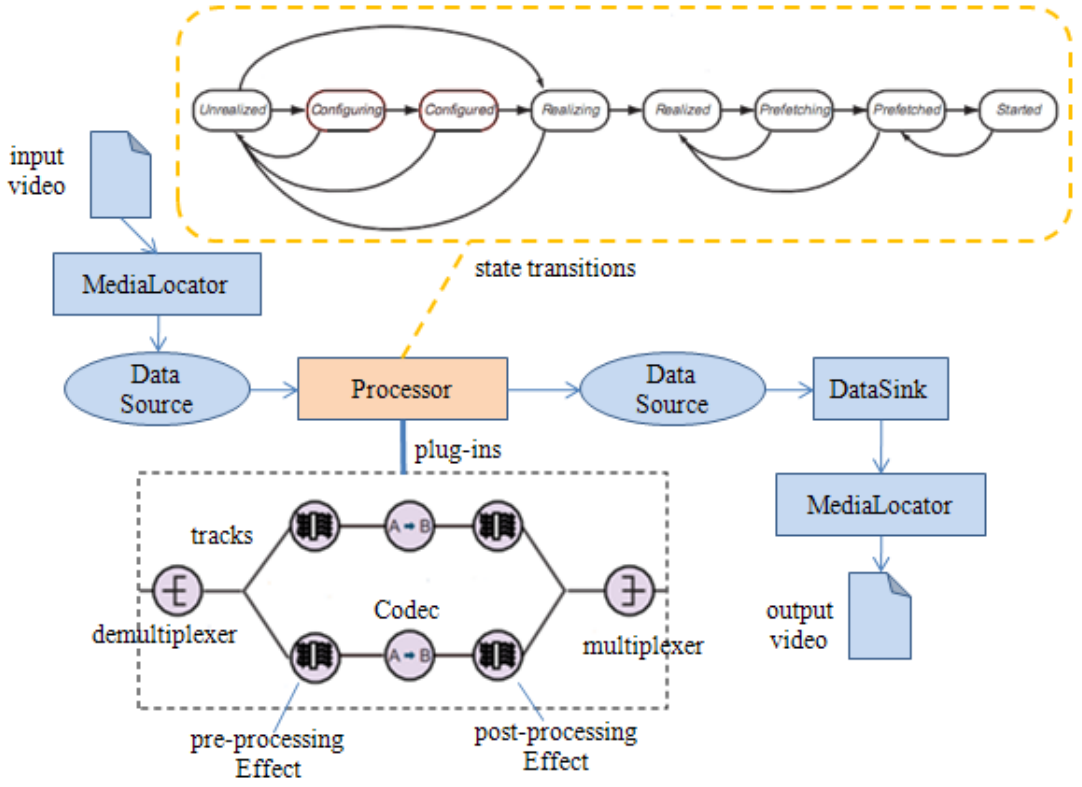

Figure 2. Processing Video in JMF.

To be fair, JMF's programming metaphor for video processing is a good one: a Processor object takes a DataSource as input, performs some user-defined processing

on the media data, and outputs the result through a DataSink to an output file. The drawbacks are in the details. Processor can move between eight different states, and typical code must wait to enter the Configured and later the Realized states before doing certain operations. This involves synchronizing with wait() and notify() calls based around events, which is tricky for most programmers to get right. In fact, the required code is almost always the same for different video processing tasks, and JMF could easily have hidden it in a more high-level version of Processor.

The Processor code typically separates the incoming data stream into audio and video tracks, so different forms of processing can be applied using JMF plug-ins. Codec plug-ins are used to decode and encode media data (see Figure 2), with the Effect class acting as a specialized Codec that performs processing other than compression or decompression. Most Effects are pre-processing tasks, carried out before the video is compressed by a Codec.

Frame-based processing is carried out inside Effect's process() method which is supplied with an input Buffer representing an incoming frame, and fills an output Buffer with the modified frame. These Buffers require a fair amount of manipulation to map them to something useful, such as byte arrays (for direct pixel manipulation) or BufferedImages (for applying drawing operations). Also, every Effect subclass requires implementations for the getSupportedInputFormats(), getSupportedOutputFormats(), setInputFormat() and setOutputFormat() methods, which are almost always the same from one Effect subclass to the next. All of this repetitive stuff could be handled by a higher-level version of Effect, relieving the programmer of the work.

On the plus side for the JMF, ia an abundance of Effect examples online, including a Rotation Effect at the JMF Solutions site (http://java.sun.com/javase/technologies/desktop/media/jmf/2.1.1/solutions/RotationEffect.html).

2. A Time Stamping Xuggler Example

The Xuggler library is available from http://www.xuggle.com/xuggler/, a website that also includes video tutorials, API documentation, and links to a xuggler-users forum (http://groups.google.com/group/xuggler-users). The library consists of two APIs: the high-level MediaTools API aimed at encoding and decoding audio and video, and the lower-level, more advanced (and complicated) Xuggler API.

MediaTools uses an event generation and listener paradigm: a media writer can be made a "listener" of a media reader, thereby receiving all the decoded data. For video modification, a custom MediaTool object is chained between the reader and writer, to change the video frame-by-frame. These components in a video time stamping example (TimeStamper.java) are shown in Figure 3.

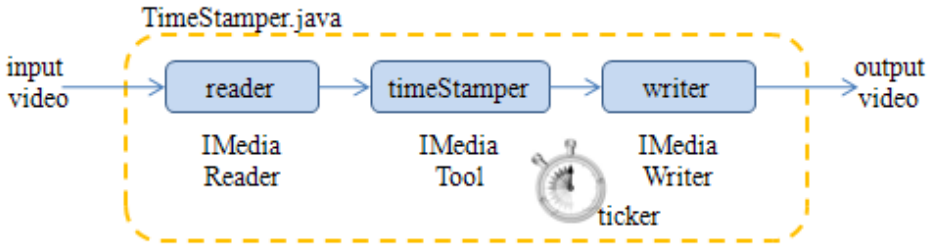

Figure 3. The Three Components of a Xuggler Video Changer.

Xuggler's IMediaReader and IMediaWriter classes hide the complexity of encoding and decoding video and audio formats. IMediaReader generates events that are sent to interested IMediaListeners (both IMediaTool and IMediaWriter implement the IMediaListener interface).

The "ticker" stopwatch in Figure 3 is a Timer task that prints a series of '."s to the screen while the video is being processed. It's unrelated to the Xuggler processing chain, and I added it to TimeStamper to give the user some simple feedback while the output was being generated.

The main() method opens the video file, sets up the tool chain, and calls IMediaReader.readPacket() to extract packets from the file and decode them. The decoded video data is automatically passed down the chain, through the media tool, to the writer.

```
public TimeStamper(String videoFnm)
{
  IMediaReader reader = initToolChain(videoFnm);
  Timer ticker = makeTicker();

  /* read and decode packets from the source file and
     then encode and write out data to the output file */
  while (reader.readPacket() == null) { }

  ticker.cancel();
  System.out.println();
}  // end of TimeStamper()
```

initToolChain() creates the chain shown in Figure 3: reader →TimeStamperTool → writer.

```
private IMediaReader initToolChain(String videoFnm)
{
  // create a media reader and configure it to generate BufferImages
  IMediaReader reader = ToolFactory.makeReader(videoFnm);
  reader.setBufferedImageTypeToGenerate(
                              BufferedImage.TYPE_3BYTE_BGR);

  // create a writer and configure it's parameters from the reader
  String outFnm = outName(videoFnm);
  IMediaWriter writer = ToolFactory.makeWriter(outFnm, reader);
```



```
  System.out.println("Writing output to " + outFnm);

  // create a tool which writes timestamps into video frames
  IMediaTool timeStamper = new TimeStamperTool();

  reader.addListener(timeStamper);
  timeStamper.addListener(writer);

  return reader;
}  // end of initToolChain()
```

The ToolFactory class handles the creation of a standard reader and writer, so I can concentrate on defining the processing class, TimeStamperTool.

The outName() method adds "Out" to the end of the filename, retaining any extension.

During debugging, it's useful to add a listener to the writer – an IMediaViewer that displays the video, on-screen statistics, and a timer:

```
// in initToolChain()
// add a viewer to the writer, to see the modified media
writer.addListener( ToolFactory.makeViewer(
           IMediaViewer.Mode.AUDIO_VIDEO, true,
           javax.swing.WindowConstants.EXIT_ON_CLOSE));
```

The availability of IMediaReader, IMediaWriter, IMediaTool, and IMediaViewer, and the chaining mechanism for listeners, makes the Xuggler library much easier to work with than the JMF. In particular, issues related to file formats are completely hidden, with the MediaTools API deciding on formats based on file extensions.

Modifying the Video

The easiest way of creating a video editing tool is to extend MediaToolAdapter, which implements IMediaTool with simple versions of the many methods that can be triggered by listener events. MediaToolAdapter also forwards events to the next object in the chain (e.g. to the writer in Figure 3).

The methods are triggered by opening and closing events (e.g. onOpen(), onClose() , onFlush()), reading and writing media packets (e.g. onReadPacket(), onWriteHeader(), onWritePacket(), onWriteTrailer()), and the detection of audio and video data (e.g. onAudioSamples(), onVideoPicture()). I only need to override onVideoPicture(), to add a timestamp to each video frame.

```
  private class TimeStamperTool extends MediaToolAdapter
  {
    public void onVideoPicture(IVideoPictureEvent event)
    {
      // get the graphics for the image
      BufferedImage im = event.getImage();
      Graphics2D g = im.createGraphics();

      IVideoPicture vidPic = event.getPicture();   // get video info
      String timeStampStr = vidPic.getFormattedTimeStamp();
```



```
      // draw yellow timestamp text
      g.setColor(Color.YELLOW);
      g.setFont(new Font("SansSerif", Font. BOLD, 32));
      g.drawString(timeStampStr, 10, im.getHeight()-10);

      super.onVideoPicture(event);     // pass event along chain
    }  // end of onVideoPicture()

  }  // end of TimeStamperTool class
```

Obtaining a BufferedImage for the incoming frame is a single operation, although the MediaReader earlier in the chain must be configured to produce the required type of BufferedImage (e.g. with/without colors, transparency). Video information, such as the frame's dimensions, type, and quality, is also easily accessed via the IVideoPicture class.

Typical frame output from TimeStamperTool is shown in Figure 4.

Figure 4. A Time Stamp in a Frame.

More MediaTools Information

Other use-cases for MediaTools are explained at http://wiki.xuggle.com/MediaTool_Introduction, including playing video, concatenating clips, building thumbnails of a media file, and converting desktop snapshots into a movie. Examples can be found at http://xuggle.googlecode.com/svn/trunk/java/xuggle-xuggler/src/com/xuggle/mediatool/demos/, and documentation at http://build.xuggle.com/view/Stable/job/xuggler_jdk5_stable/javadoc/java/api/index.html. My code is based on a Xuggler example called ModifyAudioAndVideo.java which adds a timestamp *and* adjusts the audio volume.

3. Bring on the Barcodes

Barcodes come in many varieties, roughly divided into linear (1D) and geometric patterns (2D) forms. A good summary of the main ones, including illustrations, can be found at the Wikipedia page on barcodes (http://en.wikipedia.org/wiki/Barcode). Since I want to mimic flickering video lines with barcodes, my interest lies in the linear type. One of the most popular linear barcodes is Code 128, so named because it can represent all 128 ASCII characters (http://en.wikipedia.org/wiki/Code_128),

another feature I need. A Code 128 barcode can be any length, and includes a checksum for increased reliability.

A great starting point for Java APIs for barcode generation and recognition is the barcode entry at Javapedia (http://wiki.java.net/bin/view/Javapedia/Barcode) which lists both open-source and commercial libraries. After examining the open-source APIs for their support of Code 128, I realized I would need two libraries, one to generate a barcode from text (the Barcode4J API), and another to read and decode the barcode (the ZXing API). The situation is summarized by Figure 5.

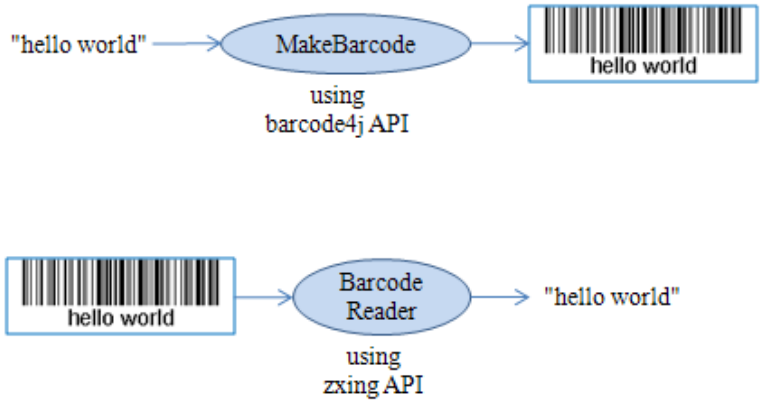

Figure 5. Generating and Reading Code 128 Barcodes.

The MakeBarcode.java and BarcodeReader.java applications are described in the next two subsections, and much of their code is reused later when I manipulate barcodes inside video.

3.1. Making a Barcode

Barcode4J can generate a wide range of 1D and 2D barcodes, and has extensions for Apache FOP, Xalan-J (for SVG output), SAXON (for XSLT), servlet usage, a command-line interface, output to SVG, EPS, Java2D/AWT, and bitmaps (http://barcode4j.sourceforge.net/). As a consequence, the Barcode4J download comes with a large set of third-party libraries, and in several different versions. Fortunately, I only need to generate a PNG file for the barcode, so don't need those extra JARs. I downloaded barcode4j-2.0-bin.zip, and extracted a *single* JAR, barcode4j-light.jar; none of the other libraries are necessary.

MakeBarcode.java reads in a string from the command line, and calls genBarcode() with the name of the output PNG file for the generated barcode.

```
// global
private static final String OUT_FNM = "codeOut.png";


public static void main(String[] args)
{
  if (args.length != 1)
    System.out.println("Usage: MakeBarcode \"message\"");
  else
```



```
      genBarcode(OUT_FNM, args[0]);
}
```

Barcode4J supports a series of barcode 'bean' classes (e.g. Code128Bean, Code39Bean, DataMatrixBean), which implement The BarcodeGenerator interface. Its generateBarcode() requires a Canvas for rendering the barcode, which is available through a range of CanvasProvider classes (e.g. BitmapCanvasProvider, EPSCanvasProvider, Java2DCanvasProvider).

genBarcode() initializes a Code128Bean object, and draws into a PNG file with a BitmapCanvasProvider.

```
// global
private static final int DPI = 150;   // dots per inch


private static void genBarcode(String fnm, String msg)
{
  Code128Bean bean = new Code128Bean();  // create a code128 barcode
  // configure the barcode generator
  bean.setModuleWidth( UnitConv.in2mm(1.0f / DPI));   // dots --> mm
  bean.setHeight(10);    // in mm
  bean.doQuietZone(true);
  bean.setQuietZone(2);  // in mm

  try {
    OutputStream out = new FileOutputStream( new File(fnm));
    BitmapCanvasProvider canvas =
        new BitmapCanvasProvider(out, "image/x-png",
                  DPI, BufferedImage.TYPE_BYTE_BINARY, false, 0);
    bean.generateBarcode(canvas, msg);

    canvas.finish();
    System.out.println("Encoded \"" + msg + "\" in " + fnm);
    out.close();
  }
  catch (Exception e)
  { System.out.println(e); }
}   // end of genBarcode()
```

A Code 128 barcode has six sections: an initial quiet zone, a start character, the data, a check character, a stop character, and a final quiet zone, as illustrated by Figure 6.

Figure 6. The Elements of a Code 128 Barcode.

Each character is made up of 11 black or white *modules*, except for the stop character which uses 13 modules. The 11 modules for a character are grouped into three bars and three spaces, which can be between 1 and 4 modules in width. The width of an individual module is set with Code128Bean.setModuleWidth(), which requires a millimeter value.

The quiet zone should be at least ten times the width of the narrowest bar or space element; for optimum hand-scanning, it should be at least 0.25 inches wide.

The barcode's height should be at least 0.15 times the barcode's length or 0.25 inches.

By default, the text of the message will be included below the barcode; this can be disabled with:

```
bean.setMsgPosition(HumanReadablePlacement.HRP_NONE);
```

It can be useful to access the generated barcode image inside the program as a BufferedImage object, which is done by calling getBufferedImage() on the canvas after the barcode has been generated but before the canvas is closed with finish(). For example:

```
bean.generateBarcode(canvas, msg);
BufferedImage im = canvas.getBufferedImage();
System.out.println("Image (w, h) (pixels): (" +
           im.getWidth() + ", " + im.getHeight() + ")");
canvas.finish();
```

3.2. Reading a Barcode

Unfortunately Barcode4j doesn't support barcode reading, and so I turned to ZXing (pronounced "zebra crossing"), an open-source, multi-format 1D/2D barcode image processing library (http://code.google.com/p/zxing/).

The library supports many platforms, including Java SE, Java ME, C#, C++, Android, the BlackBerry, and the iPhone. Fortunately, I only needed the Core and Java SE parts, which can be compiled into JAR files with Apache Ant scripts supplied with the download.

The main() function of BarcodeReader.java passes the name of the barcode image file to decode():

```
private static String decode(String fnm)
{
  BufferedImage image = null;
  try {
    image = ImageIO.read(new File(fnm));
    System.out.println("Read image from " + fnm);
  }
  catch (IOException e1) {
    System.out.println("Could not read " + fnm);
    return null;
  }

  // creating luminance source for a BufferedImage
  LuminanceSource lumSource =
        new BufferedImageLuminanceSource(image);

  // convert to binary bitmap using local thresholding
```



```
  BinaryBitmap bitmap =
        new BinaryBitmap(new HybridBinarizer(lumSource));

  Hashtable<DecodeHintType, Object> hints =
              new Hashtable<DecodeHintType, Object>();
  hints.put(DecodeHintType.TRY_HARDER, Boolean.TRUE);
                    // try for accuracy, not speed

  // barcode reader
  Code128Reader code128Reader = new Code128Reader();
  try {
    Result result = code128Reader.decode(bitmap, hints);
    return result.getText();
  }
  catch (ReaderException e) {
    System.out.println("Could not decode barcode in " + fnm);
    return null;
  }
}  // end of decode()
```

The grayscale luminance values for the loaded BufferedImage, are used to convert it into a binary bitmap. A Code 128 reader is created (ZXing contains numerous readers, and a few writers), and instructed by the DecodeHintType.TRY_HARDER hint to expend some effort to extract the string.

The result is treated as a string, but it's also possible to convert it into structured data, such as an ISBN, e-mail, SMS, or URI, with the ResultParser class:

```
Result result = code128Reader.decode(bitmap, hints);

// try decoding the barcode into a structured datatype
ParsedResult parsedResult = ResultParser.parseResult(result);
System.out.println(fnm +
           " (format: " + result.getBarcodeFormat() +
             ", type: " + parsedResult.getType() + ")");
System.out.println("Parsed result: \"" +
                   parsedResult.getDisplayResult() + "\"");

return result.getText();
```

4. Adding a Message to a Video

To increase the chance of a message being successfully extracted from the video, it's added three times, spaced at roughly equal intervals through the clip (i.e. starting at around the 25% , 50% and 75% points).

The generated barcode will be just a few pixels in height, which requires the width of each barcode module to be increased to make it easier for the decoder to find the bars and spaces. Unfortunately, the width must be increased so much that it's only possible to store a barcode made up of 2-3 characters across the width of a typically sized Flash video clip. For that reason, I decided to divide a message into individual characters, and create a separate barcode for each character.

To facilitate the correct 'gluing back together' of the characters at message extraction time, each is stored with a numerical prefix indicating its index position in the original

string. For example, "hello world" is stored as barcodes representing "0 h", "1 e", "2 l", and so on. To further simplify the decoding, a unique end-of-message character ('#') is added, and spaces are converted to '_'s. So, "hello world" is stored as "hello_world#". These steps are illustrated by Figure 7.

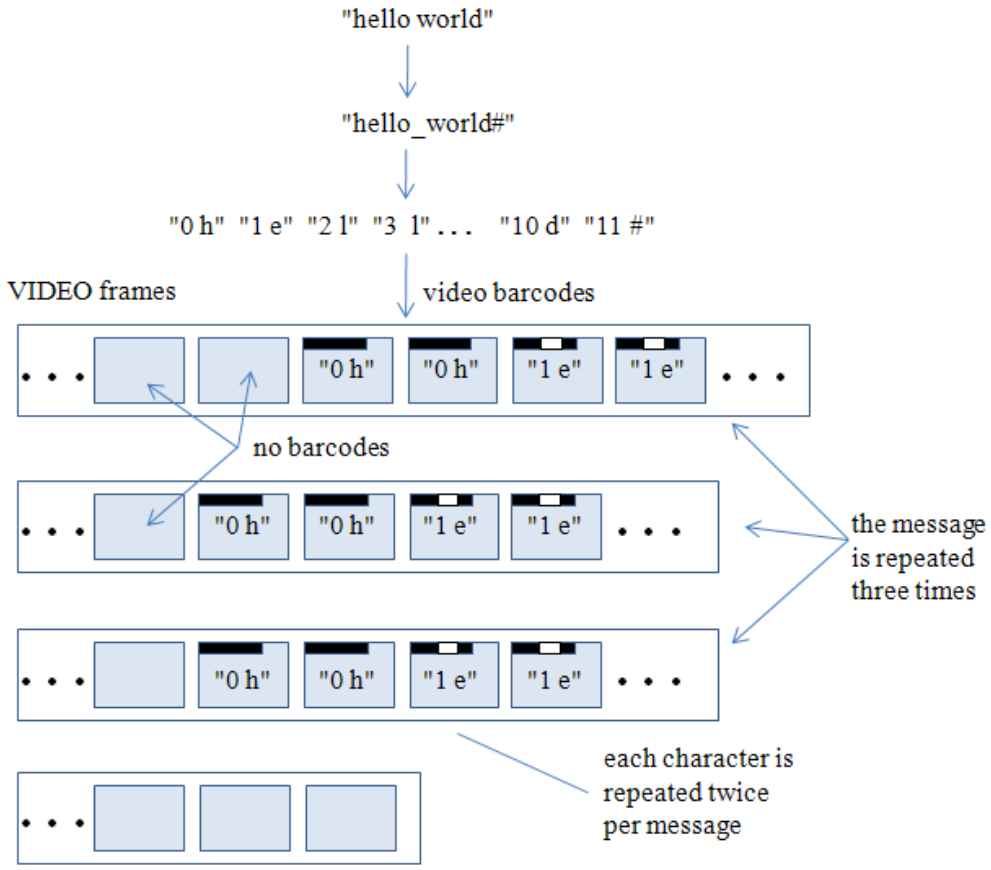

Figure 7. From Message String to Video Barcodes.

A final quirk of the encoding is that each character barcode is written to two consecutive video frames. This decreases the likelihood that video compression will remove the barcode, and makes it more likely that a character will be correctly decoded. Even if some characters are lost, the remaining message fragment will be returned, with the missing parts represented by spaces.

The message conversion to video barcodes in Figure 7 is carried out by AddVideoMsg.java, which was shown working back in Figure 1. The program's internal structure is summarized in Figure 8.

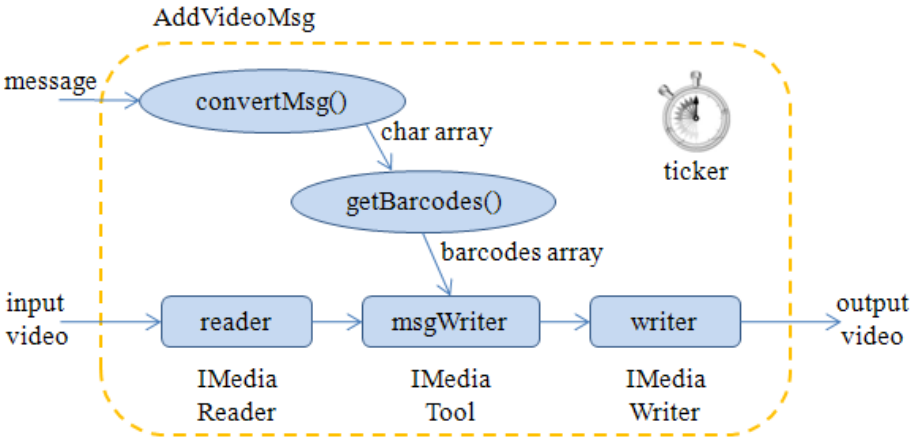

Figure 8. The Structure of AddVideoMsg.java.

The video processing sequence (reader → tool → writer) is similar to the time stamping application in Figure 3.

The AddVideoMsg() constructor creates an array of barcode images, sets up the MediaTools chain, and starts reading the video packets.

```
// globals
private static final int NUM_MSG_REPEATS = 3;
                // times the complete msg is repeated

private char[] msgBuf = null;          // stores the message
private BufferedImage[] bars = null;   // stores barcodes for message
private long msgInterval;            // time between adding a message


public AddVideoMsg(String msg, String videoFnm)
{
  msgBuf = convertMsg(msg);
  bars = genBarcodes(msgBuf);

  msgInterval = getDuration(videoFnm)/(NUM_MSG_REPEATS+1);
  System.out.println("Message interval (long): " + msgInterval);

  IMediaReader reader = initToolChain(videoFnm);

  Timer ticker = makeTicker();

  int numPackets = 0;
  while (reader.readPacket() == null)
    numPackets++;


  ticker.cancel();
  System.out.println();

  if (numPackets < msgBuf.length)
    System.out.println("Too few packets in video; lost " +
       (msgBuf.length - numPackets) + " chars of the message");
```



```
}  // end of AddVideoMsg()
```

AddVideoMsg() calculates the time spacing between the message barcodes in the video. The only unusual aspect of this is obtaining the video's running time, which is implemented in getDuration():

```
private long getDuration(String videoFnm)
{
  IContainer container = IContainer.make();
  if (container.open(videoFnm, IContainer.Type.READ, null) < 0) {
    System.out.println("Could not open: " + videoFnm);
    System.exit(1);
  }
  long duration = container.getDuration();
  System.out.println("Video duration (long): " + duration);
  container.close();
  return duration;
}  // end of getDuration()
```

An IContainer, once loaded with a video (or audio) file, can be queried for media information, such as its file size, start time, metadata, settable properties, and duration.

4.1. Generating the Barcodes

convertMsg() converts the message string into a char array, and changes certain characters (as shown in Figure 7) – spaces are replaced by '_'s and '#' is added at the end of the string. The message must be at most MAX_LEN (20) characters long, which simplifies GetVideoMsg's extraction task because it can assume an upper bound for the string's length.

genBarcodes() utilizes the Barcode4J API in a similar way to genBarcode() in the "Making a Barcode" section above (section 3.1). However, the barcodes generated by this method are very wide and short, and one is created for each character in the input array.

```
// global
private static final int DPI = 150;  // dots per inch


private BufferedImage[] genBarcodes(char[] msgBuf)
// convert each character of the message into its own barcode
{
  Code128Bean bean = new Code128Bean();   // Code 128 encoding

  // configure the barcode generator
  bean.setModuleWidth( UnitConv.in2mm(3.0f / DPI)); // wide
  bean.setHeight(4);  // short
  bean.doQuietZone(true);
  bean.setQuietZone(2);
  bean.setMsgPosition(HumanReadablePlacement.HRP_NONE);
              // no human readable text added to barcode

  BufferedImage[] bars = new BufferedImage[msgBuf.length];
  for(int i=0; i < msgBuf.length; i++) {
```



```
    try {
      BitmapCanvasProvider canvas = new BitmapCanvasProvider(
                      DPI, BufferedImage.TYPE_INT_ARGB, false, 0);
      bean.generateBarcode(canvas, "" + i + " " + msgBuf[i]);
           // add index number to front of character in the barcode
      bars[i] = fadedWhite( canvas.getBufferedImage() );
      canvas.finish();
    }
    catch (Exception e)
    { System.out.println("Failed to encode char " + i +
                                          " : " + msgBuf[i]); }
  }
  System.out.println("Pixel Height of first bar: " +
                                            bars[0].getHeight());
  return bars;
}   // end of genBarcodes()
```

genBarcodes() writes its barcode into ARGB BufferedImages, which are stored in a bars[] array. A BufferedImage has an alpha channel (the 'A' in ARGB), so the image can be given a translucent background by fadedWhite().

A Barcode4J barcode is black with a white background, but this makes it too distinctive in the video frame. fadedWhite() fixes that by searching through the image's pixels, and making all the white pixels semi-transparent. The result can be seen in the barcode in Figure 1, which blends much better with the frame, while retaining a good enough background contrast for ZXing to identify it.

```
// global color value for barcode background
private static final int WHITE = 0xFFFFFFFF;
private static final int GAUZE = 0x80FFFFFF;  // translucent white


private BufferedImage fadedWhite(BufferedImage barIm)
// convert white background of barcode to transparent gauze
{
  int w = barIm.getWidth();
  int h = barIm.getHeight();

  int[] pixels = new int[w*h];
  barIm.getRGB(0, 0, w, h, pixels, 0, w);
  for(int i=0; i < pixels.length; i++) {
    if (pixels[i] == WHITE)
      pixels[i] = GAUZE;
  }
  barIm.setRGB(0, 0, w, h, pixels, 0, w);
  return barIm;
}  // end of fadedWhite()
```

4.2. Setting up the Tool Chain

As Figure 8 shows, AddVideoMsg's tool chain consists of a reader, IMedia tool, and writer.

```
private IMediaReader initToolChain(String videoFnm)
// create a tool chain: reader -> MsgWriterTool -> writer
{
  // configure media reader to generate BufferImages
```



```
  IMediaReader reader = ToolFactory.makeReader(videoFnm);
  reader.setBufferedImageTypeToGenerate(
                          BufferedImage.TYPE_3BYTE_BGR);

  // create a writer and configure its parameters from the reader
  String outFnm = outName(videoFnm);
  IMediaWriter writer = ToolFactory.makeWriter(outFnm, reader);
  System.out.println("Writing output to " + outFnm);

  // tool writes message barcodes into the video frames
  IMediaTool msgWriter = new MsgWriterTool();

  reader.addListener(msgWriter);
  msgWriter.addListener(writer);

  return reader;
}  // end of initToolChain()
```

The only difference from the initToolChain() method in TimeStamper.java is that the video tool is an instance of MsgWriterTool.

4.3. Adding Barcodes to Video Frames

MsgWriterTool is coding in a similar way to the TimeStamperTool: it extends MediaToolAdapter and overrides onVideoPicture(). The additional complexity is due to the need to repeatedly cycle through the barcodes, and add each one twice in frame succession. Unlike TimeStamperTool, MsgWriterTool does not modify every video frame, and so requires logic to toggle image writing on and off.

```
// globals
private static final int NUM_BAR_REPEATS = 2;
               // no. of times a barcode is added per message

private int barIdx = 0;
               // counter used to cycle through the bars array
private int numAdds = 0;
               // how many times current bar has been added
private boolean isAddingMessage = false;
private long msgAddTime = msgInterval;
               // time when message should start being added


public void onVideoPicture(IVideoPictureEvent event)
// decide whether to add i-th barcode to the video frame
{
  IVideoPicture vidPic = event.getPicture();

  if (!isAddingMessage) {   // not currently adding message
    long currTime = vidPic.getTimeStamp();
    if (currTime >= msgAddTime)  // time to start adding message
      isAddingMessage = true;
  }

  if (isAddingMessage) {
    drawBarcode(barIdx, event);   // add i-th bar
    numAdds++;
    if (numAdds == NUM_BAR_REPEATS) {  // don't add i-th bar again
```



```
      barIdx = (barIdx+1)%bars.length;   // move to next barcode
      numAdds = 0;         // reset counter for next barcode
      if (barIdx == 0) {
        // back at start of message, so stop adding
        isAddingMessage = false;
        msgAddTime += msgInterval;   // calculate next add time
      }
    }
  }

  super.onVideoPicture(event);  // pass onto next tool in chain
}  // end of onVideoPicture()
```

The isAddingMessage boolean is switched to true when it's time to start adding a message (i.e. the series of character barcodes). A global index (barIdx) keeps a record of where the insertion is currently located in the series, and the numAdds variable stores how many times that current barcode has been added. The actual work of updating the video frame is done by drawBarcode().

drawBarcode() treats the video frame as a canvas on which to paint the barcode image.

```
// globals
private static final int GAUZE = 0x80FFFFFF;  // translucent white
private static final Color GAUZE_COL = new Color(GAUZE, true);


private void drawBarcode(int i, IVideoPictureEvent event)
// add i-th message char barcode to the video frame
{
  BufferedImage imVid = event.getImage();   // get graphics for frame
  int wVid = imVid.getWidth();
  int hVid = imVid.getHeight();
  Graphics2D g = imVid.createGraphics();

  BufferedImage barIm = bars[i];   // get a barcode image
  int wBar = barIm.getWidth();
  int hBar = barIm.getHeight();

  if (barIm == null)
    System.out.println("No barcode for " + i + " char '" +
                                msgBuf[i] + "'");
  else if ((wBar > wVid) || (hBar > hVid))
    System.out.println("Barcode for " + i + " char  '" +
                                msgBuf[i] + "' too large");
  else {
    g.drawImage(barIm, 0, 0, null);  // add to top-left
    g.setColor(GAUZE_COL);
         // fill rest of line after barcode with background color
    g.fillRect(wBar, 0, wVid-1, hBar);
  }
}  // end of drawBarcode()
```

Since the barcode is unlikely to extend across the entire top row of the image, any unedited space is filled with the gauze color employed by the watermark. If the watermark is too wide for the clip image, then an error message is output.

5. Extracting a Message from Video Barcodes

The input and outputs of GetVideoMsg.java are shown in Figure 9.

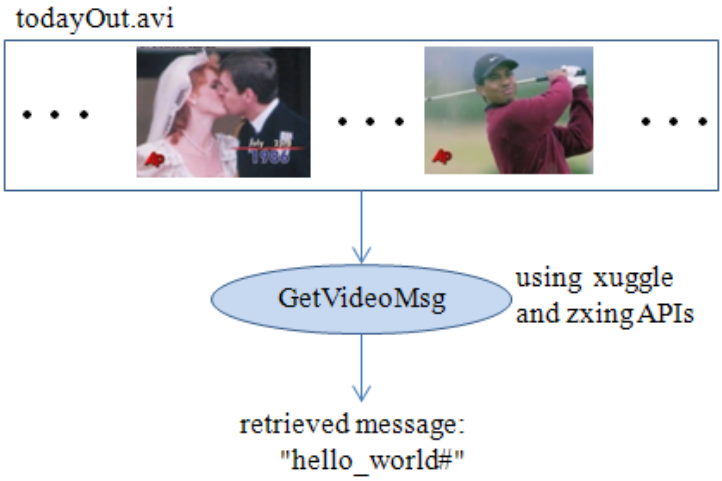

Figure 9. GetVideoMsg in Action.

GetVideoMsg may return a partial message if some characters could not be found in the video.

The important internal elements of GetVideoMsg are shown in Figure 10.

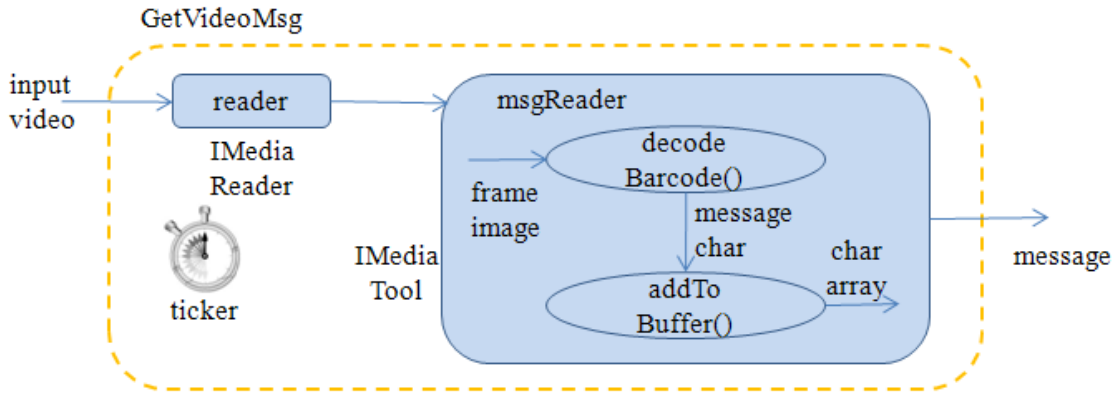

Figure 10. The Structure of GetVideoMsg.

GetVideoMsg should finish much more quickly than AddVideoMsg because it can stop as soon as it's decoded all the characters that make up the message. In my tests this mostly happened after it had encountered the first set of the character barcodes in the video, at about the 25% point. Occasionally, it needed to read some characters from the second set.

Since there's no need to save the file after extracting its message, the tool chain does not include a writer.

GetVideoMsg creates the tool chain in a similar way to AddVideoMsg, but utilizes a MsgReaderTool class, which subclasses MediaToolAdapter and overrides onVideoPicture(). It's task is to examine each video frame, looking for a barcode to translate into an index and character pair (e.g. "1 h"). The character is added to a message array at the specified index position. When the array contains the complete message, it is printed out, and GetVideoMsg finishes.

```
// in MsgReaderTool
// globals
private static final int MAX_LEN = 20;  // max length of message

private char[] msgBuf = new char[MAX_LEN+1];
                 // big enough for message and END_CHAR


public void onVideoPicture(IVideoPictureEvent event)
{
  BufferedImage im = event.getImage();  // get video frame image
  String msgPart = decodeBarcode(im);
            // get the message part stored in the barcode

  if (msgPart != null) {
    boolean isFinished = addToBuffer(msgPart);
                         // store msgPart in message buffer
    if (isFinished) {   // if message has been reconstructed
      String msg = new String(msgBuf).trim();
      System.out.println("\nCode string: \"" + msg + "\"");
      System.exit(0);
    }
  }
  super.onVideoPicture(event);  // pass to next tool in chain
}  // end of onVideoPicture()
```

A complete message may not be found, so onVideoPicture() may not terminate the search. In that case, MsgReaderTool's onClose() method will be called when the video finishes:

```
public void onClose(ICloseEvent event)
// print out the (partial) message
{
  String msg = new String(msgBuf).trim();
  System.out.println("\nCode string at close: \"" + msg + "\"");
}  // end of onClose()
```

5.1. Decoding a Barcode

The ZXing API is capable of searching an image for a barcode, and then decoding it. However, I decided to speed things up, and make the search more reliable, by giving ZXing just the clipped top row of the video frame where the barcode should be located. The rest of decodeBarcode() is similar to the decode() method in the "Reading a Barcode" section (section 3.2):

```
// globals
private static final int BARCODE_HEIGHT = 2;
        // the pixel height of a barcode image
```



```
private String decodeBarcode(BufferedImage image)
// extract message part from barcode stored in image
{
  if (image == null) {
    System.out.println("No image found");
    return null;
  }

  // get clip containing barcode (the top rows of the image)
  BufferedImage clipIm = null;
  try {
    clipIm = image.getSubimage(0, 0,
                  image.getWidth(), BARCODE_HEIGHT);
  }
  catch(RasterFormatException e) {
    System.out.println("Could not clip image");
    return null;
  }

  // creating luminance source for a BufferedImage
  LuminanceSource lumSource =
               new BufferedImageLuminanceSource(clipIm);
  // convert to a binary bitmap using local thresholding
  BinaryBitmap bitmap = new BinaryBitmap(
                          new HybridBinarizer(lumSource));

  Hashtable<DecodeHintType, Object> hints =
               new Hashtable<DecodeHintType, Object>();
  hints.put(DecodeHintType.TRY_HARDER, Boolean.TRUE);
               // try for accuracy, not speed

  // decode barcode
  Code128Reader code128Reader = new Code128Reader();
  Result result = null;
  try {
    result = code128Reader.decode(bitmap, hints);
    return result.getText();
  }
  catch (ReaderException e) {
    return null;
  }
}  // end of decodeBarcode()
```

decodeBarcode() returns null if it fails to find a barcode, or can't decode it.

addToBuffer() is passed a message part (which has the format "<number> <character>") and attempts to add the character to that position in a char array. The size of the complete message will be known when it's END_CHAR ('#') is read in, and this is employed along with a counter (numCharsFound) to determine when all of the message has been read.

```
// globals
private static final char END_CHAR = '#';  // final char of message
private static final int MAX_LEN = 20;     // max length of message

private char[] msgBuf = new char[MAX_LEN+1];
                              // for message and the END_CHAR
```



```
private int numCharsFound = 0;  // no of chars found for msgBuf[]
private int endPosn = -1;       // location of END_CHAR in msgBuf[]


private boolean addToBuffer(String msgPart)
{
  // extract position and character from msgPart
  String[] toks = msgPart.split(" ");
  if (toks.length != 2) {
    System.out.println("Tokenization of \"" + msgPart + "\" failed");
    return false;
  }

  int posn = -1;
  try {
    posn = Integer.parseInt(toks[0]);
  }
  catch(NumberFormatException e) {
    System.out.println("\"" + toks[0] + "\" is not an integer");
    return false;
  }

  if (toks[1].length() != 1) {
    System.out.println("\"" + toks[1] + "\" is not a single char");
    return false;
  }
  char ch = toks[1].charAt(0);

  // store msg character in msgBuf[] (maybe)
  if ((posn < 0) || (posn >= msgBuf.length)) {
    System.out.println("Position value: " + posn + " out of range");
    return false;
  }

  if (msgBuf[posn] == ' ') {   // that posn is currently empty
    msgBuf[posn] = ch;    // fill it
    numCharsFound++;
  }

  // record END_CHAR position in the buffer
  if (msgBuf[posn] == END_CHAR)
    endPosn = posn;

  if (endPosn != -1) {  // we know the end position
    if (numCharsFound-1 == endPosn)   // so check if array is full
      return true;
  }
  return false;
}  // end of addToBuffer()
```

addToBuffer() doesn't make any assumptions about the ordering of the message parts it receives, so it would be possible to disperse the barcodes throughout the video in any order.

