Magnetic Stripe Reading

Joseph Battaglia
sephail@sephail.net http://www.sephail.net

Originally appearing in *2600 Magazine*, Spring 2005

Introduction

Good magnetic stripe readers are hard to come by. Most are expensive, only capable of reading one or two tracks, and have inconvenient interfaces. In this article I will describe the process of making an extremely cheap, simple, and reliable single-track reader from parts that are readily available. We will be interfacing the reader to the microphone input of a sound card, which is very convenient for use with most laptops and desktops.

I will not be discussing the theory and concepts of magnetic stripe technology and the assumption is made that you are somewhat familiar with the topic. For a simplistic overview of magnetic stripe technology that is easy to read and understand, I recommend that you read the classic article "Card-O-Rama: Magnetic Stripe Technology and Beyond" by Count Zero, which can be found quickly by doing a web search for keywords in the title.

Materials

Below is a list of materials you'll need to construct the reader.

- Magnetic head
  Magnetic heads are extremely common. Discarded cassette tape players contain magnetic heads of almost the exact size needed (the small difference won't matter for our application). Simply obtain a discarded cassette tape player and remove the magnetic head without damaging it. These heads are usually secured with one or two screws which can be useful when building the reader, so don't discard them.
- 3.5mm mono phone plug (with 2-conductor wire)
  You can find this on a discarded monaural earphone or in an electronics store.
- Soldering iron with solder

Optional:

- Wood (or other sturdy material) base to mount magnetic head
- Ruler or other straight edge to slide cards on

Construction

The actual hardware design is incredibly simple. The interface consists of simply connecting the output of the magnetic head directly to the mic input of a sound card. Solder the wire connecting the 3.5mm mono phone plug (base and tip) to the leads of the magnetic stripe head. Polarity does not matter.

I recommend that you mount the head in a way that makes it easy to swipe a card over it with a constant velocity. This is where your custom hardware ingenuity comes in. Mount a ruler (or other straight edge) perpendicular to the magnetic head, with the reading solenoid (usually visible as a

black rectangle on the head) at the correct distance from the base for the corresponding track. Track 1 starts at 0.223" from the bottom of the card, Track 2 starts at 0.333", and Track 3 starts at 0.443".

Alternatively, you can purchase a surplus reader with no interface (i.e., scrapped or with a cheap TTL interface) and follow the same instructions with the exception that the magnetic head will already be mounted. Most surplus readers come preset to Track 2, although it is usually a simple hardware mod to move it to the track you'd like to read. This will save you the trouble of building a custom swiping mechanism and will also improve the reliability of the reads. There are surplus readers that can be purchased for less than $10 US at various online merchants.

Software

In this project, the software does all the heavy lifting. The "dab" utility included in this article takes the raw DSP data from your sound card, decodes the FSK (frequency shift keying - a.k.a. Atkin Biphase) modulation from the magnetic stripe, and outputs the binary data. Additionally, you can decode the binary data using the "dmsb" utility to output the ASCII characters and perform an LRC check to verify the integrity of the data, provided that the stripe conforms to the specifications described in ISO 7811, 7813, and optionally ISO 4909 (for the uncommon Track 3). Becoming familiar with these specifications will help you understand the contents of the magnetic stripe when viewing the decoded data.

The provided software is more proof-of-concept than production code, and should be treated as such. That said, it does its job well. It is open source and released under the MIT license. Feel free to contribute.

Requirements

- Linux (or the desire to port to another operating system)
- A configured 16-bit sound card
- Access to the /dev/dsp device
- libsndfile

Note that "dab" can also take input from any audio file supported by libsndfile. However, it must be a clean sample that starts at the beginning of the file. This is useful to eliminate the requirement of a sound card and allow samples to be recorded from another device (e.g., an MP3 player/recorder) and decoded at another time.

Obtaining

dab.c (v0.7)- Decode Atkin Biphase

dmsb.c (v0.1)- Decode (standard) Magnetic Stripe Binary

Code is available from the Code Listing appendices and http://www.sephail.net/articles/magstripe

Compiling

Edit any configuration #defines near the top of the dab.c file and proceed to compile the source with the following commands:

```
cc dab.c -o dab -lsndfile
cc dmsb.c -o dmsb
```

Usage

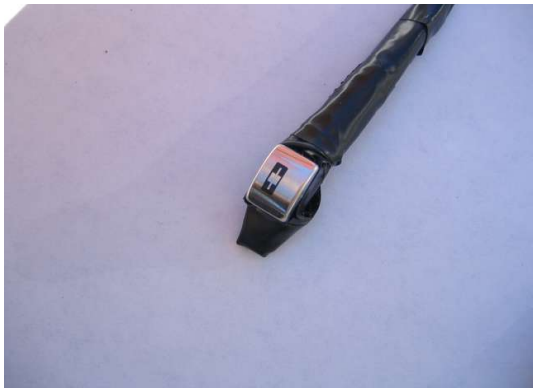
```
Usage: dab [OPTIONS]

  -a,  --auto-thres   Set auto-thres percentage
                      (default: 30)
  -d,  --device       Device to read audio data from
                      (default: /dev/dsp)
  -f,  --file         File to read audio data from
                      (use instead of -d)
  -h,  --help         Print help information
  -m,  --max-level    Shows the maximum level
                      (use to determine threshold)
  -s,  --silent       No verbose messages
  -t,  --threshold    Set silence threshold
                      (default: automatic detect)
  -v,  --version      Print version information

Usage: dmsb [OPTIONS]

  -V,  --verbose      Verbose messages

  -h,  --help         Print help information
  -v,  --version      Print version information

dmsb will wait on stdin for raw magnetic stripe data (string of 0s and 1s
followed by a newline) and print the decoded data to stdout.
```

Be sure that the mic is set as the recording device for your sound card (using a utility such as aumix or your preferred mixer). Standard usage on the command line with the hardware interfaced directly to the sound card (mic in) will be as follows with standard cards:

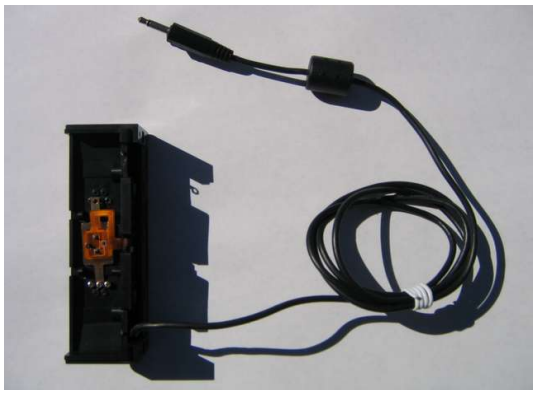
```
./dab | ./dmsb
```

Pictures

My original reader. With this reader I would use a ruler as a track guide. This way I could not only read the three standard tracks, but also data on non-standard cards, some of which have tracks in odd positions such as through the middle of the card.



My current reader, made of a modified surplus reader which is only capable of reading the three standard tracks.



Examples

Below are some examples of a few (hopefully) less common cards as to get an idea of the sort of data you're likely to find.

Park Inn (Berlin-Alexanderplatz) Door Key Cards

```
Room: 2006
Checkout Date: 12/30/2004
Card 1
Track 2 Data: ;5101152006010912130124000120000000000?
Card 2
Track 2 Data: ;5101152006020912130124000120000000000?

Room: 2005
Checkout Date: 12/30/2004
Card 1
Track 2 Data: ;5101152005010160230124000120000000000?
Card 2
Track 2 Data: ;5101152005020160230124000120000000000?
```

SEPTA Monthly TransPass Cards

```
Month: November 2004
Serial: 001467
Track 2 Data: ;0101001101041130040000001467?

Month: June 2003
Serial: 002421
Track 2 Data: ;0101000601030630030000002421?

Month: January 2002
Serial: 028813
Track 2 Data: ;0101000101020131020000028813?
```

Sony Connect Cash Cards

```
Card Number: 603571 010462 1134569
PIN: 9014
Track 1 Data: %B6035710104621134569^^49120000040?
Track 2 Data: ;6035710104621134569=49120000040?
```

```
Card Number: 603571 010462 1132282
PIN: 5969
Track 1 Data: %B6035710104621132282^^49120008147?
Track 2 Data: ;6035710104621132282=49120008147?
```

Starbucks Cards

```
Card Number: 6015 0613 2715 8426
Track 1 Data: %B6010565061327158^0040/MOMSDAY04^25010004000060018426           ?
Track 2 Data: ;6010565061327158=25010004000060018426?

Card Number: 6014 5421 5637 9529
Track 1 Data: %B6010564542156377^0027/EXCLUSIVEB2B04^25010004000060019529           ?
Track 2 Data: ;6010564542156377=25010004000060019529?

Card Number: 6014 5421 6302 5757
Track 1 Data: %B6010564542156377^0027/EXCLUSIVEB2B04^25010004000060019529           ?
Track 2 Data: ;6010564542163027=25010004000060015757?
```

Conclusion

This project was originally started for the New York City MetroCard decoding project that you may have heard about on *Off The Hook*. Nearly all commercial readers are unable to dump the raw data as it exists on the MetroCard and, even if they could, they are priced way above our (and most hobbyists') budget limitations. This solution has worked very well for us and can aid you in reverse-engineering cards that you may have as well. The "dmsb" application available online can be used for simply decoding standard cards that you have laying around as well.

While my construction example demonstrates a fairly straightforward and typical use of a magnetic stripe reader, many other uses can be considered.

For instance, since all the data obtained from the reader itself is audio, the device can be interfaced to a digital audio recording device, such as one of the many MP3 (and other codec) player/recorders on the market. You could then set the device to record, interfaced the same way with the magnetic stripe reader, and have a stand-alone reader small enough to fit in your pocket. Later, you'd view and edit the captured audio file, saving the clean waveform to a standard .wav file to be analyzed with "dab" (which, in fact, has this capability). You can even construct the reader in an inconspicuous way, so onlookers would never realize the device's capability.

How is this significant? Reading boarding passes with magnetic stripes is a perfect application. These are generally only available in the waiting area of airports. They're issued at check-in and collected when you board, leaving a very small time margin during which the stripe can be scanned. In my case, I had been flagged for additional security and the infamous "SSSS" was printed on my pass. Using my reader, I was able to duck into a bathroom and quickly read the data into my mp3 player/recorder for later analysis. (I discovered a mysterious code on track 2 (normally blank) which read: "C 13190-2******" as well as an "S" at the end of the passenger data on track 1.)

But there are other more sinister applications. What if one of the waiters at your favorite restaurant built this device and swiped the card of everyone who pays with credit? From the data obtained, an exact clone of the credit card could be created. Credit card fraud would quickly become out of control if this were commonplace.

The same principle could be applied to reverse-engineering an unknown magnetic stripe technology. While individual card samples are often much more difficult to obtain, scanning samples as you obtain them enables you to gather samples at an astonishing rate. This way, supporters can loan you cards to scan on the spot. I have personally used this method for the

MetroCard decoding project and it works extremely well.

I could go on and on with more examples of the implications of this sort of design, but I'd like to hear back from the readers as to what other ideas may have been thought up. All feedback is appreciated and, time permitting, all questions will be answered.

Hopefully this project makes you realize how certain types of technology are priced way above what they have to be to keep them away from "us" because of the fear of malicious use. I also hope it encourages more projects like this to surface so we can learn about and use technology without the restrictions imposed upon us by big corporations.

Code Listing (dab.c)

```
/* dab.c - Decode Aiken Biphase

   Copyright (c) 2004-2005 Joseph Battaglia <sephail@sephail.net>

   Code contributions / patches:
     Mike Castleman <mlc@2600.com>
     Ed Wandasiewicz <wanded@breathemail.net>

   Permission is hereby granted, free of charge, to any person obtaining a copy
   of this software and associated documentation files (the "Software"), to
   deal in the Software without restriction, including without limitation the
   rights to use, copy, modify, merge, publish, distribute, sublicense, and/or
   sell copies of the Software, and to permit persons to whom the Software is
   furnished to do so, subject to the following conditions:

   The above copyright notice and this permission notice shall be included in
   all copies or substantial portions of the Software.

   THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
   IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
   FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
   AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
   LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
   FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS
   IN THE SOFTWARE.

   Changelog:
     0.1 (Sep 2004):
         'audiomag' released
     0.2 (Oct 2004):
         2600 MetroCard decoding project started
         changed name from 'audiomag' to 'dab'
         now requires only one "clocking" bit
         optimized for reading non-standard cards (eg. MetroCards)
     0.3 (Nov 2004):
         improved decoding algorithm
         added max_level functionality
     0.4 (Dec 2004):
         fixed bug when calculating threshold from percentage
     0.5 (Dec 2004):
         improved decoding algorithm
         improved automatic threshold detection
         added support for reading from a file with libsndfile (Mike C.)
     0.6 (Jan 2005):
         fixed broken flags
         improved libsndfile use
     0.7 (Aug 2005):
         fixed potential segmentation fault (Ed W.)

   Compiling:
     cc dab.c -o dab -lsndfile
*/


#include <fcntl.h>
#include <getopt.h>
#include <sndfile.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ioctl.h>
#include <sys/soundcard.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

/*** defaults ***/
#define DEVICE        "/dev/dsp" /* default sound card device */
#define SAMPLE_RATE   192000     /* default sample rate (hz) */
#define SILENCE_THRES 5000       /* initial silence threshold */
/*** end defaults ***/

/* #define DISABLE_VC */

#define AUTO_THRES    30    /* pct of highest value to set silence_thres to */
#define BUF_SIZE      1024  /* buffer size */
#define END_LENGTH    200   /* msec of silence to determine end of sample */
#define FREQ_THRES    60    /* frequency threshold (pct) */
#define MAX_TERM      60    /* sec before termination of print_max_level() */
#define VERSION       "0.7" /* version */


short int *sample = NULL;
int sample_size = 0;



/********** function wrappers **********/

/* allocate memory with out of memory checking
   [size]          allocate size bytes
   returns         pointer to allocated memory */
void *xmalloc(size_t size)
{
  void *ptr;

  ptr = malloc(size);
  if (ptr == NULL) {
    fprintf(stderr, "Out of memory.\n");
    exit(EXIT_FAILURE);
  }

  return ptr;
}


/* reallocate memory with out of memory checking
   [ptr]           memory to reallocate
   [size]          allocate size bytes
   returns         pointer to reallocated memory */
void *xrealloc(void *ptr, size_t size)
{
  void *nptr;

  nptr = realloc(ptr, size);
  if (nptr == NULL) {
    fprintf(stderr, "Out of memory.\n");
    exit(EXIT_FAILURE);
  }

  return nptr;
}


/* copy a string with out of memory checking
   [string]         string to copy
   returns          newly allocated copy of string */
char *xstrdup(char *string)
{
  char *ptr;

  ptr = xmalloc(strlen(string) + 1);
  strcpy(ptr, string);

  return ptr;
}


/* read with error checking
   [fd]            file descriptor to read from
   [buf]           buffer
   [count]         bytes to read
   returns         bytes read */
ssize_t xread(int fd, void *buf, size_t count)
{
  int retval;

  retval = read(fd, buf, count);
  if (retval == -1) {
    perror("read()");
    exit(EXIT_FAILURE);
  }

  return retval;
}

/********** end function wrappers **********/



/********** version functions **********/

/* prints version
   [stream]        output stream */
void print_version(FILE *stream)
{
  fprintf(stream, "dab - Decode Aiken Biphase\n");
  fprintf(stream, "Version %s\n", VERSION);
  fprintf(stream, "Copyright (c) 2004-2005 ");
  fprintf(stream, "Joseph Battaglia <sephail@sephail.net>\n");
}


/* prints version and help
   [stream]        output stream
   [exec]          string containing the name of the program executable */
void print_help(FILE *stream, char *exec)
{
  print_version(stream);
  fprintf(stream, "\nUsage: %s [OPTIONS]\n\n", exec);
  fprintf(stream, "  -a,  --auto-thres   Set auto-thres percentage\n");
  fprintf(stream, "                      (default: %d)\n", AUTO_THRES);
  fprintf(stream, "  -d,  --device       Device to read audio data from\n");
  fprintf(stream, "                      (default: %s)\n", DEVICE);
  fprintf(stream, "  -f,  --file         File to read audio data from\n");
  fprintf(stream, "                      (use instead of -d)\n");
  fprintf(stream, "  -h,  --help         Print help information\n");
  fprintf(stream, "  -m,  --max-level    Shows the maximum level\n");
  fprintf(stream, "                      (use to determine threshold)\n");
  fprintf(stream, "  -s,  --silent       No verbose messages\n");
  fprintf(stream, "  -t,  --threshold    Set silence threshold\n");
  fprintf(stream, "                      (default: automatic detect)\n");
  fprintf(stream, "  -v,  --version      Print version information\n");
}

/********** end version functions **********/



/********** dsp functions **********/

/* sets the device parameters
   [fd]            file descriptor to set ioctls on
   [verbose]       prints verbose messages if true
   returns         sample rate */
int dsp_init(int fd, int verbose)
{
  int ch, fmt, sr;

  if (verbose)
    fprintf(stderr, "*** Setting audio device parameters:\n");

  /* set audio format */
  if (verbose)
    fprintf(stderr, "    Format: AFMT_S16_LE\n");
  fmt = AFMT_S16_LE;
  if (ioctl(fd, SNDCTL_DSP_SETFMT, &fmt) == -1) {
    perror("SNDCTL_DSP_SETFMT");
    exit(EXIT_FAILURE);
  }
  if (fmt != AFMT_S16_LE) {
    fprintf(stderr, "*** Error: Device does not support AFMT_S16_LE\n");
    exit(EXIT_FAILURE);
  }

  /* set audio channels */
  if (verbose)
    fprintf(stderr, "    Channels: 1\n");
  ch = 0;
  if (ioctl(fd, SNDCTL_DSP_STEREO, &ch) == -1) {
    perror("SNDCTL_DSP_STEREO");
    exit(EXIT_FAILURE);
  }
  if (ch != 0) {
```

```
    fprintf(stderr, "*** Error: Device does not support monaural recording\n");
    exit(EXIT_FAILURE);
  }

  /* set sample rate */
  if (verbose)
    fprintf(stderr, "    Sample rate: %d\n", SAMPLE_RATE);
  sr = SAMPLE_RATE;
  if (ioctl(fd, SNDCTL_DSP_SPEED, &sr) == -1) {
    perror("SNDCTL_DSP_SPEED");
    exit(EXIT_FAILURE);
  }
  if (sr != SAMPLE_RATE)
    fprintf(stderr, "*** Warning: Highest supported sample rate is %d\n", sr);

  return sr;
}


/* prints the maximum dsp level to aid in setting the silence threshold
   [fd]           file descriptor to read from
   [sample_rate]   sample rate of device */
void print_max_level(int fd, int sample_rate)
{
  int i;
  short int buf, last = 0;

  printf("Terminating after %d seconds...\n", MAX_TERM);

  for (i = 0; i < sample_rate * MAX_TERM; i++) {

    /* read from fd */
    xread(fd, &buf, sizeof (short int));

    /* take absolute value */
    if (buf < 0)
      buf = -buf;

    /* print if highest level */
    if (buf > last) {
      printf("Maximum level: %d\r", buf);
      fflush(stdout);
      last = buf;
    }
  }

  printf("\n");
}


/* finds the maximum value in sample
   ** global **
   [sample]        sample
   [sample_size]   number of frames in sample */
short int evaluate_max(void)
{
  int i;
  short int max = 0;

  for (i = 0; i < sample_size; i++) {
    if (sample[i] > max)
      max = sample[i];
  }

  return max;
}


/* pauses until the dsp level is above the silence threshold
   [fd]           file descriptor to read from
   [silence_thres] silence threshold */
void silence_pause(int fd, int silence_thres)
{
  short int buf = 0;

  /* loop while silent */
  while (buf < silence_thres) {

    /* read from fd */
    xread(fd, &buf, sizeof (short int));

    /* absolute value */
    if (buf < 0)
      buf = -buf;
  }
}


/* gets a sample, terminating when the input goes below the silence threshold
   [fd]           file descriptor to read from
   [sample_rate]   sample rate of device
   [silence_thres] silence threshold
   ** global **
   [sample]        sample
   [sample_size]   number of frames in sample */
void get_dsp(int fd, int sample_rate, int silence_thres)
{
  int count = 0, eos = 0, i;
  short buf;

  sample_size = 0;

  /* wait for sample */
  silence_pause(fd, silence_thres);

  while (!eos) {
    /* fill buffer */
    sample = xrealloc(sample, sizeof (short int) * (BUF_SIZE * (count + 1)));
    for (i = 0; i < BUF_SIZE; i++) {
      xread(fd, &buf, sizeof (short int));
      sample[i + (count * BUF_SIZE)] = buf;
    }
    count++;
    sample_size = count * BUF_SIZE;

    /* check for silence */
    eos = 1;
    if (sample_size > (sample_rate * END_LENGTH) / 1000) {
      for (i = 0; i < (sample_rate * END_LENGTH) / 1000; i++)  {
        buf = sample[(count * BUF_SIZE) - i - 1];
        if (buf < 0)
          buf = -buf;
        if (buf > silence_thres)
          eos = 0;
      }
    } else
      eos = 0;
  }
}

/********** end dsp functions **********/



/********** begin sndfile functions **********/

/* open the file
   [fd]          file to open
   [verbose]     verbosity flag
   ** global **
   [sample_size] number of frames in the file */
SNDFILE *sndfile_init(int fd, int verbose)
{
  SNDFILE *sndfile;
  SF_INFO sfinfo;

  /* clear sfinfo structure */
  memset(&sfinfo, 0, sizeof(sfinfo));

  /* set sndfile from file descriptor */
  sndfile = sf_open_fd(fd, SFM_READ, &sfinfo, 0);
  if (sndfile == NULL) {
    fprintf(stderr, "*** Error: sf_open_fd() failed\n");
    exit(EXIT_FAILURE);
  }

  /* print some statistics */
  if (verbose) {
    fprintf(stderr, "*** Input file format:\n"
            "    Frames: %i\n"
            "    Sample Rate: %i\n"
            "    Channels: %i\n"
            "    Format: 0x%08x\n"
            "    Sections: %i\n"
            "    Seekable: %i\n",
            (int)sfinfo.frames, sfinfo.samplerate, sfinfo.channels,
            sfinfo.format, sfinfo.sections, sfinfo.seekable);
  }

  /* ensure that the file is monaural */
  if (sfinfo.channels != 1) {
    fprintf(stderr, "*** Error: Only monaural files are supported\n");
    exit(EXIT_FAILURE);
  }

  /* set sample size */
  sample_size = sfinfo.frames;

  return sndfile;
}


/* read in data from libsndfile
   [sndfile]     SNDFILE pointer from sf_open() or sf_open_fd()
   ** global **
   [sample]      sample
   [sample_size] number of frames in sample */
void get_sndfile(SNDFILE *sndfile)
{
  sf_count_t count;

  /* allocate memory for sample */
  sample = xmalloc(sizeof(short int) * sample_size);

  /* read in sample */
  count = sf_read_short(sndfile, sample, sample_size);
  if (count != sample_size) {
    fprintf(stderr, "*** Warning: expected %i frames, read %i.\n",
            sample_size, (int)count);
    sample_size = count;
  }
}

/********** end sndfile functions **********/



/* decodes aiken biphase and prints binary
   [freq_thres]    frequency threshold
   ** global **
   [sample]        sample
   [sample_size]   number of frames in sample */
void decode_aiken_biphase(int freq_thres, int silence_thres)
{
  int i = 0, peak = 0, ppeak = 0;
  int *peaks = NULL, peaks_size = 0;
  int zerobl;

  /* absolute value */
  for (i = 0; i < sample_size; i++)
    if (sample[i] < 0)
      sample[i] = -sample[i];

  /* store peak differences */
  i = 0;
  while (i < sample_size) {
    /* old peak value */
    ppeak = peak;
    /* find peaks */
    while (i < sample_size && sample[i] <= silence_thres)
      i++;
    peak = 0;
    while (i < sample_size && sample[i] > silence_thres) {
      if (sample[i] > sample[peak])
        peak = i;
      i++;
    }
    if (peak - ppeak > 0) {
      peaks = xrealloc(peaks, sizeof(int) * (peaks_size + 1));
      peaks[peaks_size] = peak - ppeak;
      peaks_size++;
    }
  }

  /* decode aiken biphase allowing for
     frequency deviation based on freq_thres */
```

```
  /* ignore first two peaks and last peak */
  if (peaks_size < 2) {
    fprintf(stderr, "*** Error: No data detected\n");
    exit(EXIT_FAILURE);
  }
  zerobl = peaks[2];
  for (i = 2; i < peaks_size - 1; i++) {
    if (peaks[i] < ((zerobl / 2) + (freq_thres * (zerobl / 2) / 100)) &&
        peaks[i] > ((zerobl / 2) - (freq_thres * (zerobl / 2) / 100))) {
      if (peaks[i + 1] < ((zerobl / 2) + (freq_thres * (zerobl / 2) / 100)) &&
          peaks[i + 1] > ((zerobl / 2) - (freq_thres * (zerobl / 2) / 100))) {
        printf("1");
        zerobl = peaks[i] * 2;
        i++;
      }
    } else if (peaks[i] < (zerobl + (freq_thres * zerobl / 100)) &&
               peaks[i] > (zerobl - (freq_thres * zerobl / 100))) {
      printf("0");
#ifndef DISABLE_VC
      zerobl = peaks[i];
#endif
    }
  }
  printf("\n");

}



/* main */
int main(int argc, char *argv[])
{
  int fd;
  SNDFILE *sndfile = NULL;

  /* configuration variables */
  char *filename = NULL;
  int auto_thres = AUTO_THRES, max_level = 0, use_sndfile = 0, verbose = 1;
  int sample_rate = SAMPLE_RATE, silence_thres = SILENCE_THRES;

  /* getopt variables */
  int ch, option_index;
  static struct option long_options[] = {
    {"auto-thres",   0, 0, 'a'},
    {"device",       1, 0, 'd'},
    {"file",         1, 0, 'f'},
    {"help",         0, 0, 'h'},
    {"max-level",    0, 0, 'm'},
    {"silent",       0, 0, 's'},
    {"threshold",    1, 0, 't'},
    {"version",      0, 0, 'v'},
    { 0,             0, 0,  0 }
  };

  /* process command line arguments */
  while (1) {

    ch = getopt_long(argc, argv, "a:d:f:hmst:v", long_options, &option_index);

    if (ch == -1)
      break;

    switch (ch) {
      /* auto-thres */
      case 'a':
        auto_thres = atoi(optarg);
        break;
      /* device */
      case 'd':
        filename = xstrdup(optarg);
        break;
      /* file */
      case 'f':
        filename = xstrdup(optarg);
        use_sndfile = 1;
        break;
      /* help */
      case 'h':
        print_help(stdout, argv[0]);
        exit(EXIT_SUCCESS);
        break;
      /* max-level */
      case 'm':
        max_level = 1;
        break;
      /* silent */
      case 's':
        verbose = 0;
        break;
      /* threshold */
      case 't':
        auto_thres = 0;
        silence_thres = atoi(optarg);
        break;
      /* version */
      case 'v':
        print_version(stdout);
        exit(EXIT_SUCCESS);
        break;
      /* default */
      default:
        print_help(stderr, argv[0]);
        exit(EXIT_FAILURE);
        break;
    }
  }

  /* print version */
  if (verbose) {
    print_version(stderr);
    fprintf(stderr, "\n");
  }

  /* check for incorrect use of command-line arguments */
  if (use_sndfile && max_level) {
    fprintf(stderr, "*** Error: -f and -m switches do not mix!\n");
    exit(EXIT_FAILURE);
  }

  /* set default if no device is specified */
  if (filename == NULL)
    filename = xstrdup(DEVICE);

  /* open device for reading */
  if (verbose)
    fprintf(stderr, "*** Opening %s\n", filename);
  fd = open(filename, O_RDONLY);
  if (fd == -1) {
    perror("open()");
    exit(EXIT_FAILURE);
  }

  /* open sndfile or set device parameters */
  if (use_sndfile)
    sndfile = sndfile_init(fd, verbose);
  else
    sample_rate = dsp_init(fd, verbose);

  /* show user maximum dsp level */
  if (max_level) {
    print_max_level(fd, sample_rate);
    exit(EXIT_SUCCESS);
  }

  /* silence_thres sanity check */
  if (!silence_thres) {
    fprintf(stderr, "*** Error: Invalid silence threshold\n");
    exit(EXIT_FAILURE);
  }

  /* read sample */
  if (use_sndfile)
    get_sndfile(sndfile);
  else {
    if (verbose)
      fprintf(stderr, "*** Waiting for sample...\n");
    get_dsp(fd, sample_rate, silence_thres);
  }

  /* automatically set threshold */
  if (auto_thres)
    silence_thres = auto_thres * evaluate_max() / 100;

  /* print silence threshold */
  if (verbose)
    fprintf(stderr, "*** Silence threshold: %d (%d%% of max)\n",
            silence_thres, auto_thres);

  /* decode aiken biphase */
  decode_aiken_biphase(FREQ_THRES, silence_thres);

  /* close file */
  close(fd);

  /* free memory */
  free(sample);

  exit(EXIT_SUCCESS);

  return 0;
}
/* end dab.c */
```

Code Listing (dmsb.c)

```
/* dmsb.c - Decodes (standard) Magnetic Stripe Binary

   Copyright (c) 2004 Joseph Battaglia <sephail@sephail.net>

   Permission is hereby granted, free of charge, to any person obtaining a copy
   of this software and associated documentation files (the "Software"), to
   deal in the Software without restriction, including without limitation the
   rights to use, copy, modify, merge, publish, distribute, sublicense, and/or
   sell copies of the Software, and to permit persons to whom the Software is
   furnished to do so, subject to the following conditions:

   The above copyright notice and this permission notice shall be included in
   all copies or substantial portions of the Software.

   THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
   IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
   FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
   AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
   LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
   FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS
   IN THE SOFTWARE.

   Compiling:
     cc dmsb.c -o dmsb
*/


#define BUF_SIZE 2048
#define VERSION "0.1"

#include <getopt.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>



/********** function wrappers **********/

/* allocate memory with out of memory checking
   [size]          allocate size bytes
   returns         pointer to allocated memory */
void *xmalloc(size_t size)
{
  void *ptr;

  ptr = malloc(size);
  if (ptr == NULL) {
    fprintf(stderr, "Out of memory.\n");
    exit(EXIT_FAILURE);
  }

  return ptr;
}


/* reallocate memory with out of memory checking
   [ptr]           memory to reallocate
   [size]          allocate size bytes
   returns         pointer to reallocated memory */
void *xrealloc(void *ptr, size_t size)
{
  void *nptr;

  nptr = realloc(ptr, size);
  if (nptr == NULL) {
    fprintf(stderr, "Out of memory.\n");
    exit(EXIT_FAILURE);
  }

  return nptr;
}


/* copy a string with out of memory checking
   [string]         string to copy
   returns          newly allocated copy of string */
char *xstrdup(char *string)
{
  char *ptr;

  ptr = xmalloc(strlen(string) + 1);
  strcpy(ptr, string);

  return ptr;
}

/********** end function wrappers **********/



/********** version functions **********/

/* print version information
   [stream]        output stream */
void print_version(FILE *stream)
{
  fprintf(stream, "dmsb - Decode (standard) Magnetic Stripe Binary\n");
  fprintf(stream, "Version %s\n", VERSION);
  fprintf(stream, "Copyright (c) 2004 Joseph Battaglia
<sephail@sephail.net>\n");
}


/* print help information
   [stream]        output stream
   [exec]          string containing the name of the program executable */
void print_help(FILE *stream, char *exec)
{
  print_version(stream);
  fprintf(stream, "\nUsage: %s [OPTIONS]\n", exec);
  fprintf(stream, "\n");
  fprintf(stream, "  -V,  --verbose      Verbose messages\n");
  fprintf(stream, "\n");
  fprintf(stream, "  -h,  --help         Print help information\n");
  fprintf(stream, "  -v,  --version      Print version information\n");
  fprintf(stream, "\n");
  fprintf(stream, "dmsb will wait on stdin for raw magnetic stripe ");
  fprintf(stream, "data (string of 0s and 1s\n");
  fprintf(stream, "followed by a newline) and print the decoded data to ");
  fprintf(stream, "stdout.\n");
}

/********** end version functions **********/



/********** string functions **********/

/* returns a pointer to the reversed string
   [string]        string to reverse
   returns         newly allocated reversed string */
char *reverse_string(char *string)
{
  char *rstring;
  int i, string_len;

  string_len = strlen(string); /* record string length */

  /* allocate memory for rstring */
  rstring = xmalloc(string_len + 1);

  for (i = 0; i < string_len; i++) /* reverse string and store in rstring */
    rstring[i] = string[string_len - i - 1];

  rstring[string_len] = '\0'; /* terminate rstring */

  return rstring; /* return rstring */
}

/********** end string functions **********/



/********** parsing functions **********/

/* parse ABA format raw bits and return a pointer to the decoded string
   [bitstring]     string to decode
   returns         decoded string */
char *parse_ABA(char *bitstring)
{
  char *decoded_string, *lrc_start, *start_decode, *string;
  char lrc[] = {1, 1, 0, 1, 0}; /* initial condition is LRC of the start
                                   sentinel */
  int asciichr, charcnt = 0, i, j;

  /* make a copy of bitstring and store it in string */
  string = xstrdup(bitstring);

  /* look for start sentinel */
  if ((start_decode = strstr(string, "11010")) == NULL) {
    free(string); /* free string memory */
    return NULL; /* could not find start sentinel */
  }

  /* set start_decode to first bit (start of first byte) after start
     sentinel */
  start_decode += 5;

  /* look for end sentinel */
  if ((lrc_start = strstr(string, "11111")) == NULL) {
    free(string); /* free string memory */
    return NULL; /* could not find end sentinel */
  }
  /* must be a multiple of 5 */
  while ((strlen(start_decode) - strlen(lrc_start)) % 5) /* search again */
    if ((lrc_start = strstr(++lrc_start, "11111")) == NULL) {
      free(string); /* free string memory */
      return NULL; /* cound not find end sentinel */
    }

  lrc_start[0] = '\0'; /* terminate start_decode at end sentinel */

  lrc_start += 5; /* set the pointer to the LRC */
  if (lrc_start[5] != '\0') /* terminate LRC if not already */
    lrc_start[5] = '\0';

  /* allocate memory for decoded_string */
  decoded_string = xmalloc((strlen(start_decode) / 5) + 3);

  decoded_string[charcnt++] = ';'; /* add start sentinel */

  /* decode each set of bits, check parity, check LRC, and add to
     decoded_string */
  while (strlen(start_decode)) {

    for (i = 0, j = 0; i < 4; i++) /* check parity */
      if (start_decode[i] == '1')
        j++;
    if (((j % 2) && start_decode[4] == '1') ||
        (!(j % 2) && start_decode[4] == '0')) {
      free(string); /* free string memory */
      free(decoded_string); /* free decoded_string memory */
      return NULL; /* failed parity check */
    }

    asciichr = 48; /* generate ascii value from bits */
    asciichr += start_decode[0] == '1' ? 1 : 0;
    asciichr += start_decode[1] == '1' ? 2 : 0;
    asciichr += start_decode[2] == '1' ? 4 : 0;
    asciichr += start_decode[3] == '1' ? 8 : 0;

    decoded_string[charcnt++] = asciichr; /* add character to decoded_string */

    for (i = 0; i < 4; i++) /* calculate LRC */
      lrc[i] = lrc[i] ^ (start_decode[i] == '1') ? 1 : 0;

    start_decode += 5; /* increment start_decode to next byte */
  }

  decoded_string[charcnt++] = '?'; /* add end sentinel */
  decoded_string[charcnt] = '\0'; /* terminate decoded_string */

  for (i = 0; i < 4; i++) /* calculate CRC of end sentinel */
```

```
    lrc[i] = lrc[i] ^ 1;

  for (i = 0, j = 0; i < 4; i++) /* set LRC parity bit */
    if (lrc[i])
      j++;
  if (!(j % 2))
    lrc[4] = 1;
  else
    lrc[4] = 0;

  for (i = 0; i < 5; i++) /* check CRC */
    if ((lrc[i] && lrc_start[i] == '0') ||
        (!lrc[i] && lrc_start[i] == '1')) {
      free(string); /* free string memory */
      free(decoded_string); /* free decoded_string memory */
      return NULL; /* failed CRC check */
    }

  free(string); /* free string memory */
  return decoded_string;
}


/* parse IATA format raw bits and return a pointer to the decoded string
   [bitstring]     string to decode
   returns         decoded string */
char *parse_IATA(char *bitstring)
{
  char *decoded_string, *lrc_start, *start_decode, *string;
  char lrc[] = {1, 0, 1, 0, 0, 0, 1}; /* initial condition is LRC of the start
                                         sentinel */
  int asciichr, charcnt = 0, i, j;

  /* make a copy of bitstring and store it in string */
  string = xstrdup(bitstring);

  /* look for start sentinel */
  if ((start_decode = strstr(string, "1010001")) == NULL) {
    free(string); /* free string memory */
    return NULL; /* could not find start sentinel */
  }

  /* set start_decode to first bit (start of first byte) after start
     sentinel */
  start_decode += 7;

  /* look for end sentinel */
  if ((lrc_start = strstr(string, "1111100")) == NULL) {
    free(string); /* free string memory */
    return NULL; /* could not find end sentinel */
  }
  /* must be a multiple of 7 */
  while ((strlen(start_decode) - strlen(lrc_start)) % 7)
    /* search again */
    if ((lrc_start = strstr(++lrc_start, "1111100")) == NULL) {
      free(string); /* free string memory */
      return NULL; /* cound not find end sentinel */
    }

  lrc_start[0] = '\0'; /* terminate start_decode at end sentinel */

  lrc_start += 7; /* set the pointer to the LRC */
  if (lrc_start[7] != '\0') /* terminate LRC if not already */
    lrc_start[7] = '\0';

  /* allocate memory for decoded_string */
  decoded_string = xmalloc((strlen(start_decode) / 7) + 3);

  decoded_string[charcnt++] = '%'; /* add start sentinel */

  /* decode each set of bits, check parity, check LRC, and add to
     decoded_string */
  while (strlen(start_decode)) {

    for (i = 0, j = 0; i < 6; i++) /* check parity */
      if (start_decode[i] == '1')
        j++;
    if (((j % 2) && start_decode[6] == '1') ||
        (!(j % 2) && start_decode[6] == '0')) {
      free(string); /* free string memory */
      free(decoded_string); /* free decoded_string memory */
      return NULL; /* failed parity check */
    }

    asciichr = 32; /* generate ascii value from bits */
    asciichr += start_decode[0] == '1' ? 1  : 0;
    asciichr += start_decode[1] == '1' ? 2  : 0;
    asciichr += start_decode[2] == '1' ? 4  : 0;
    asciichr += start_decode[3] == '1' ? 8  : 0;
    asciichr += start_decode[4] == '1' ? 16 : 0;
    asciichr += start_decode[5] == '1' ? 32 : 0;

    decoded_string[charcnt++] = asciichr; /* add character to decoded_string */

    for (i = 0; i < 6; i++) /* calculate LRC */
      lrc[i] = lrc[i] ^ (start_decode[i] == '1') ? 1 : 0;

    start_decode += 7; /* increment start_decode to next byte */
  }

  decoded_string[charcnt++] = '?'; /* add end sentinel */
  decoded_string[charcnt] = '\0'; /* terminate decoded_string */

  for (i = 0; i < 5; i++) /* calculate CRC of end sentinel */
    lrc[i] = lrc[i] ^ 1;
  lrc[5] = lrc[5] ^ 0;

  for (i = 0, j = 0; i < 6; i++) /* set LRC parity bit */
    if (lrc[i])
      j++;
  if (!(j % 2))
    lrc[6] = 1;
  else
    lrc[6] = 0;

  for (i = 0; i < 7; i++) /* check CRC */
    if ((lrc[i] && lrc_start[i] == '0') ||
        (!lrc[i] && lrc_start[i] == '1')) {
      free(string); /* free string memory */
      free(decoded_string); /* free decoded_string memory */
      return NULL; /* failed CRC check */
    }

  free(string); /* free string memory */
  return decoded_string;

}

/********** end parsing functions **********/


int main(int argc, char *argv[])
{
  char buf[BUF_SIZE], *rbuf, *decoded_data;
  int verbose = 0;
  int ch, option_index;

  static struct option long_options[] = {
    {"verbose", 0, 0, 'V'},
    {"help"   , 0, 0, 'h'},
    {"version", 0, 0, 'v'},
    { 0       , 0, 0,  0 }
  };

  while ((ch = getopt_long(argc, argv, "Vhv", long_options, &option_index))
         != -1) {
    switch (ch)
    {
      case 'V': /* verbose */
        verbose = 1;
        break;
      case 'h': /* help */
        print_help(stdout, argv[0]);
        exit(EXIT_SUCCESS);
        break;
      case 'v': /* version */
        print_version(stdout);
        exit(EXIT_SUCCESS);
        break;
      default: /* invalid option */
        print_help(stderr, argv[0]);
        exit(EXIT_FAILURE);
        break;
    }
  }

  if (verbose) {
    print_version(stderr);
    fprintf(stderr, "Waiting for data on stdin...\n");
  }

  fgets(buf, BUF_SIZE, stdin); /* get string from stdin */

  if (verbose) {
    fprintf(stderr, "Trying to decode using ABA...");
    fflush(stderr);
  }


  if ((decoded_data = parse_ABA(buf)) != NULL) { /* try ABA */
    if (verbose) {
      fprintf(stderr, "success\n");
      fprintf(stderr, "ABA format detected:\n");
    }
    printf("%s\n", decoded_data); /* print decoded data */
    exit(EXIT_SUCCESS);
  }

  if (verbose) {
    fprintf(stderr, "reversing bits...");
    fflush(stderr);
  }

  rbuf = reverse_string(buf); /* reverse string and try again */

  if ((decoded_data = parse_ABA(rbuf)) != NULL) { /* try ABA */
    if (verbose) {
      fprintf(stderr, "success\n");
      fprintf(stderr, "ABA format detected (bits reversed):\n");
    }
    printf("%s\n", decoded_data);
    exit(EXIT_SUCCESS);
  }

  if (verbose)
    fprintf(stderr, "failed\n");

  if (verbose) {
    fprintf(stderr, "Trying to decode using IATA...");
    fflush(stderr);
  }


  if ((decoded_data = parse_IATA(buf)) != NULL) { /* try IATA */
    if (verbose) {
      fprintf(stderr, "success\n");
      fprintf(stderr, "IATA format detected:\n");
    }
    printf("%s\n", decoded_data); /* print decoded data */
    exit(EXIT_SUCCESS);
  }

  if (verbose) {
    fprintf(stderr, "reversing bits...");
    fflush(stderr);
  }

  if ((decoded_data = parse_IATA(rbuf)) != NULL) { /* try IATA with reverse */
    if (verbose) {
      fprintf(stderr, "success\n");
      fprintf(stderr, "IATA format detected (bits reversed):\n");
    }
    printf("%s\n", decoded_data);
    exit(EXIT_SUCCESS);
  }

  if (verbose)
    fprintf(stderr, "failed\n");

  printf("Detection failed\n");

  exit(EXIT_FAILURE);

  return 0;
}
/* end dmsb.c */
```