Biphase-Mark encoding and decoding - a short
guide.

Neil Barnes - nailed_barnacle@hotmail.com
1/12/2007

Abstract
Biphase-Mark encoding is sometimes recommended as a communication method to get data from one place to another. This guide is intended to show how it works, how to code it and decode it, and how to recognise it.

Contents
1 What is it?  2
2 The software  2
3 Synchronous vs. Asynchronous  3
4 RIFF files  4
5 Coding and decoding  5
6 Biphase  5
7 Unbiphase  6
8 Performance  7
9 Implementing a decoder on AVR  9
10 Conclusion  10
11 The source code  10
  11.1 Makefile .................................................. 10
  11.2 biphase.c .................................................. 10
  11.3 unbiphase.c ............................................... 14
  11.4 biphase for AVR ......................................... 19
1 What is it?

It’s a method of encoding data onto either an audio signal, or on a logic-level signal path. Its characteristics include:
- self-clocking; you don’t need to know the data rate to decode it
- average DC is zero; capacitively coupled audio signals will still decode properly
- robust; it depends on phase-inversions to identify bits
- able to handle either synchronous or asynchronous data
- able to manage a wide range of data rates
- able to be safely decoded on changing data rates
  - e.g. from a spooling tape

The signal is defined such that there is a phase change in the level of the signal at the end of each bit. If the bit is a one, then there is a further phase change in the centre of the bit period. So, a continuing stream of zeros at a 1kbit/s data rate would be a 500Hz signal; a continuing stream of ones would be a 1kHz signal. Normal data would have a mixture of both; it has a very distinctive sound.

Biphase-mark encoding is widely used in the broadcasting industry for the distribution and recording of linear time-code; the audio signal is recorded on analogue video or audio tapes and can be read both during normal playback and while spooling quickly. It’s also often distributed around studio areas as a source of time-of-day data; a reader can be plugged into a jack and the time can be immediately displayed.

Another surprisingly common place to find biphase-mark encoding is in the SP-DIF digital connection (and of course the closely related AES-EBU digital audio).

2 The software

In the attached files, I have included the source code for an encoder - biphase.c - and a decoder - unbiphase.c. Rather than try and provide an example for AVR - where it can be implemented in so many different ways, depending on the type of AVR chosen, resources required by other parts of the program, and so on - I have provided generic C programs for a PC. They are written for GCC under Linux though they should work with minimal changes - if any - on any of the MS C compilers. The only point to note is that they require variables of int=32 bits, short=16 bits, and char = 8 bits.

Caveat

It should be noted that neither of these programs is robust enough for production software. Both have far too little error checking, particularly when opening, reading from, or writing to files; neither behave properly in respect of the Microsoft RIFF format files.
3 Synchronous vs. Asynchronous

Synchronous data is presented as a continuous bitstream; immediately after one byte of data follows the next. There are no start or stop bits (though parity may be included in the bitstream), so some other mechanism must be used to synchronise the start of the data. Also, if the data input stops, the bitstream must continue. This is generally managed by having a 'sync byte' in the bitstream which is output whenever there is no 'real' data to output. The sync byte must be something which is unambiguous and cannot be mistaken for the real data; this therefore reduces the number of data byte options available in the main data - for example, on an eight-bit system, the code '0xFF' might be reserved as a sync byte. However, if there is the possibility of entering the data stream at a random position, you have to ensure that the data itself in any random position cannot simulate the sync byte - for example, the sequence '0x0F', '0xF0' would look like a sync byte.

Linear Time Code - as mentioned above - is designed to be read not only at multiple and varying speeds, but also forwards and backwards. To achieve this capability, a sixteen-bit sync block is included which cannot occur as a legal bit pattern elsewhere in the 80-bit packet, which is repeated every 25th of a second. The pattern in the sync block can be identified to give the direction of the data.

Asynchronous data is most familiar to most as the non-return to zero RS232 style data. The data can occur at any time, which implies an idle state on the line. On the RS232 format (ignoring the actual transmission levels) this is a '1'. To mark that data is starting, a start bit is used - a zero - followed by a number of data bits. After an optional parity bit, one or more stop bits - '1's - will restore the line to the idle level. As a result, each 8-bit byte of data takes a minimum of ten bit periods to transmit. Because the start bit can occur at random times, the line has to be sampled somewhat faster than the nominal data rate; normally, sixteen times. This enables the following bits to be sampled somewhere near their centre point for improved noise performance.

A disadvantage of such data is that the clock rates of the transmitter and receiver must be close. Once the two rates differ by around 5%, the sampling point of the last bit can slip to the previous or following bit, producing an error. This is a common problem with AVRs clocked on their internal uncalibrated clocks; either the clock must be calibrated in software or crystal or other high-accuracy clocks must be used. But because the bit timing is self-clocked on the biphase-mark coding, clock drift is not important; the data is guaranteed.

After some consideration, I decided to implement a non-synchronous format for my example. The main driver for this is that most people are more familiar with the format, and it provides a more complex example - though not as complex as a finished implementation might be.

My format - when writing to audio - therefore has a short period of '1's as an idle state, followed by the data itself in the one start bit, eight data bits (bit 0 first), no parity bit, and one stop bit form. After the last data, another short period of '1's is recorded - unnecessary in this instance but with a more complex system with say short frames of data containing some sort of error correction,
providing an ideal inter-block idle state.

The audio format is mono WAVE format, 16,000 bits per second, sixteen bits per sample. Since I am recording a square wave, this is excessive but means I can perform other unpleasantries on the audio data - filtration, compression etc - with standard tools.

4 RIFF files

RIFF files are a Microsoft standard packaging stream for - usually - multimedia storage. They may contain audio, video, random data, and metadata, contained in chunks. The WAVE format is reserved for audio applications.

A chunk consists of a four-byte ascii header with the name of the chunk, four bytes (unsigned long) to give the length of the chunk data, and then the chunk data itself. A chunk can contain sub-chunks to any number or level, though it is unusual to get any deeper than three levels. Note that the WAVE ident does not have an immediate data length following it. Although MS refer to this as a chunk, my feeling is that it should be considered as part of the data - so the chunk consists of the WAVE identifier and a minimum of two subchunks - the fmt and data chunks.

The start of a minimal WAVE file looks like:

RIFF <- the riff chunk ident
xxxx <- the length of the RIFF chunk (usually the entire remainder of the file)
WAVE <- the WAVE chunk ident
fmt <- the format chunk identifier
xxxx <- format chunk length
.. <- data depends on the length and type of the format chunk
data <- the data chunk identifier
xxxx <- data chunk length
.. <- many many data samples

The fmt chunk data describes the format of the data (PCM, MP3, or other compression), mono or stereo, bit rate, sample rate and so on. The audio data needs to be understood in the terms of this header. In this case, we’re going for a straight-forward PCM mono signal at sixteen bits; hard to get wrong.

I should stress that the method I have used to create the file is not recommended by Microsoft (or me, really) in that all I do is preload the chunk structure as a single ‘waveheader’ structure. This produces a legal file, and I haven’t come across any software which can’t read it, but modern wave structures include extra fields. The file reader I provide in uniphase.c can cheerfully read this format, but may break on other wave files - though it will not crash, you may find that you’re treating the extra fields (or indeed, extra chunks) as audio data.
The official way to deal with these files is to use Microsoft’s RIFF API; creating chunks on demand to write and 'walking the chunks' to find the chunks in which you are interested and ignoring the rest. However, that option is not immediately available in the GCC libraries, and would distract from the point of the examples.

5 Coding and decoding

Coding biphase mark data is simple; one needs to arrange a phase change at the bit rate and another phase change in the middle of that period, for the '1'. How that is arranged is up to the programmer. In this version, I’ve just used an integer number of audio samples to time the phase change. Alternate approaches might include timer interrupts every half a bit period, with a routine that decides whether or not a phase change is required at any particular interrupt. It can also be derived in hardware, using a couple of flip-flops.

Decoding is also reasonably simple, depending on whether you know the data rate and that it will be stable. If you do know it, then it’s as easy as looking at the interval since the last phase change. If it’s more than 3/4 of the bit period, you’ve just received a zero; otherwise, you got half of a one. If you already had the first half, this is the second half and you can return a one; otherwise, flag this as a first-half received. Note that because you don’t necessarily know the phasing of an arbitrary stream of ones, it’s possible that the following half-cycle might be a zero; this unambiguously identifies the zero but data prior to that must be considered suspect.

If you don’t know the bit rate, it’s a little more complex. Now, you need not only the period of the current half-cycle but that of the previous half-cycle. If these two are the same (within 0.75 and 1.5 times) then we can safely assume that we have another of what we previously had... but we might not know yet what it is. If the current period is more than 1.5 times the previous, then we have definitely received a zero. If instead, it is less than 0.75 times the previous period, then we have definitely received the first half of a one. Only after such an event can we unambiguously decode the continuing bit stream. Once the bit stream is identified, ones and zeros will follow automatically, even if the data rate changes, provided the change is less than approximately one sixth of a bit period per bit.

6 Biphase

Biphase requires two parameters: the name of the file to be encoded, and the file for the output data. It will produce a (quite large) audio file as described above. You will need to provide the complete filename; I recommend .wav as the suffix. The input file can be anything; text, executables, whatever. Note that the output file will cheerfully overwrite anything already existing with the same filename.
Most of the code is actually concerned with managing the creation of the audio file; creating the file, setting the header values, writing the header, and then after the data is written going back to the header to fill in the file sizes.

The biphase coding section is only three subroutine - and that for clarity... write_1() and write_0() make a fast cycle or a slow half-cycle respectively; 'last’ holds the value written by a subroutine on its last half-cycle (and returned by the routine) so that the phase is maintained between calls. write_byte() packages the a given byte by wrapping it in a start and stop bit, and then calling write_1() or write_0() as appropriate to create the audio data. It also passes the 'last’ value back. Everything is wrapped up in a simple routine that reads bytes from the input file and calls write_byte() for each byte.

Sequential bytes follow with no idle time other than the stop bit which is always a one. This gives the fastest possible transmission time for the selected bit rate. The bit rate itself can be modified quite a lot; provided there’s at least one sample at each level between transition times it can be decoded - so with the audio package as described, it will work as fast as 8kb/s. Of course, if you are using a direct connection between processor, the signal can be arbitrarily fast, provided you have sufficient time to decode. On a 16MHz AVR I would expect (I haven’t tried!) to be able to read at least half a megabit per second.

That’s pretty much it... simple!

7 Unbiphase

Unbiphase also expects two parameters: a mono 16-bit wave file - the bit rate is unimportant provided that the data is recorded at 1000 bits per second - and a filename to write the decoded output. It will overwrite any existing file of the same name.

The main part of the program looks after opening the audio file. As discussed above, this is not a standard way of writing RIFF files but it suffices for the purposes of this demonstration. The wave header information is read and sufficient checks are made to ensure that the file is of a suitable type. Because we have a defined bit rate, we can use the simpler of the methods discussed above to decode the signal - we have access to the sample rate in the wave header we can easily calculate how many samples mark the necessary periods.

Decoding from a stored audio signal is simple since the timing information can be extracted effectively instantly; on something like the ACR the timing can
be done several ways. If actual audio is being decoded, then the ADC complete interrupt can be used as a regular tick, or a logic-level signal can be used to drive an interrupt pin. An internal timer/counter is stored after every interrupt, and reset for the next period (the interrupt sense can be inverted after every interrupt).

The main routine simply calls `get_byte()` while there are still audio samples remaining.

8 Performance

Having shown that the data can be successfully encoded and decoded, I decided to see how much abuse the signal could take before it fell over.

The original audio file `test.wav` was created by encoding the source code for `biphase.c`; all further processing was done within Audacity under Kubuntu Linux using the standard defaults. There are a touch under ten thousand characters in the original text file.

'diff' was used to check the accuracy of the transcription; the audio after processing was exported from Audacity as a .wav file and then decoded using `unbiphase`.

Remember that these are all being decoded with the basic version of the code, with simplistic zero crossing/edge detection.

**Test 1 - no processing (test.wav)**

On inspection, the audio has a classic square wave signal shape. As expected, the file produced no change between the input and the output.

**Test 2 - Low Pass (testlopass.wav)**

This Audacity process - with the corner frequency set at 993Hz - shows classic exponential rise-times of the square wave after filtration. However, the decoded audio once again had no errors.

**Test 3 - High Pass (testhipass.wav)**

This Audacity process results, as expected, in a file full of spikes at each bit edge and little DC component; however, once again, the result decodes perfectly.

Since the coding survives basic frequency response changes successfully, let’s have a look at more extreme changes to the signal. In all these cases, we’re deliberately inducing distortions in the signal. In these tests, we use Audacity to speed up and slow down the signal, changing the pitch. I don’t know the method used by Audacity, but the normal translation method involves oversampling the original signal, digital filtration to the desired bandwidth, and resampling the resulting file. So not only do we have data edges arriving at the wrong speed,
but they’re likely to have added jitter - since they’re now unlikely to be at exact integer multiples of samples. On top of that, we won’t have exact square waves any more - as an examination of the waveform will soon show.

Test 4 - Speeding it up (testfast15.wav)

Well, well, well... it can cope with this; no errors on decoding. At 25% (test file not included) it doesn’t work at all well, though. I had expected it to cope with 25% fast, and I believe the errors are rather to do with the extra jitter and sample position rather than the limits of the decoding - remember that we are still using the simple decoder which is expecting a fixed data rate.

Test 5 - slowing it down (testslo w25.wav)

And sure enough, once again there are no errors. Much slower than this, though, and this version will fail. However, using an adaptive decoder, where the previous bit periods are used to detect one/zero differences.

Ok, we’ve demonstrated that it can cope with a fair amount of mangling; what happens if we try compressing the signal?

Well, it happens that zip or rar will actually compress the original un-meddled-with square-wave signal very well indeed; it only contains two different values for the audio signal. However, more general audio compression methods are psycho-acoustic models; rather than trying to reproduce the original waveform they endeavour to produce a signal which the ear will accept as similar to the original. In this process, both the phase of the signal components, and some of the original components themselves, are lost.

Therefore, we might reasonably expect the data to be somewhat mangled after it’s been through this type of perceptual encoding. In this experiment, I treat it to both mp3 coding and ogg coding. Using the default parameters for Audacity, the mp3 file is compressed at a lower bitrate than the ogg file; the original 2.3MB file compresses to about 220kB in mp3, and 400kB in ogg.

In both cases, we compress the original audio using Audacity, close the file, reopen the resultant .ogg or .mp3 file, and export the result as a .wav file - so this has tested both the compression and decompression paths.

Note that the attachment regime on AVR Freaks does not allow posting of ogg or mpg files directly; also, if I try and rar the decoded wav files they end up the same size as the original, so instead I have renamed the original ogg and mp3 files with a .txt suffix; remove this and save before decoding.

Test 6 - MP3 - the people’s favourite

A curious result. What we see is that the data has failed horribly - but not as might be expected. Instead of a complete failure, or randomly correct bytes throughout the file, we find that it appears to have worked in short bursts - first a few dozen random garbage characters, then a few dozen error free. I don’t
know why this should be, but I believe it may be due to the way the mp3 codec
manages its framing gain stages.

Test 7 - OGG - license free compression

But here’s a surprising result - a completely error free output. To be fair,
the compression is not as severe as the mp3 compression, and back-to-back
subjective listening tests have shown a better sound quality than the same bit-
rate as mp3 on the same source material.

However, it seems that if one wanted - for example - to control using a spare
audio channel, compressed .ogg files are the way to go.

9 Implementing a decoder on AVR

So, finally, a short application to allow the AVR to use biphase mark coded
signals. I should stress that this has not been built on real hardware; it has,
however, been tested under emulation/simulation on the WinAVR platform with
a stimulus file created in a manner similar to the code above. The code and the
associated stimulus file are attached below.

In this application, aimed at the ATMega8 running at 8MHz, the logic-level
data is applied at the PD2 pin - INT0. The output is piped directly to PD0 -
TX - and is presented as a bit-by-bit copy. Since the original code is at 1000
bits per second, the output is perfectly valid RS232 at 1000 baud. The start
and stop bits are present in the original coding.

I’d recommend - for playing - using the audio output from a PC to deliver
the code; use an op-amp or comparator to detect the zero crossings and provide
a logic-level signal to the input pin.

The code is in two parts.

In main, the direction is set for Port D. We enable interrupts on INT0 on
any change of logic level (which provides interrupts on every edge) and finally
enable interrupts globally. We set the timer1 - having calculated that a count
of about 4,000 or 8,000 clocks will occur between edges, timer 1 is the right size,
though it would be simple to use, for example, one of the eight bit counters with
a prescale of 64. Then, we just sit in a loop and amuse ourselves with anything
else going on.

In the interrupt section, we decode in a very similar way - but structurally
simpler - to the PC examples earlier. We can’t hang around in the interrupt
waiting for the second half of a ‘1’ so instead, we hold a static flag that re-
members when we’ve had the first half. Rather than building the data byte in
memory, in this case we just spit it straight out of the TX pin, as each bit is
decoded. Nothing to stop *you* managing the code, of course...
10 Conclusion

So, here it is. I think I’ve demonstrated that biphase mark encoding is a versatile, robust, and easy method of moving data or commands around. It’s easy to decode on an AVR (there are only fifteen lines of actual code in the routine described above!) and uses very little processor time. It will work over a huge range of speeds, and over very long distances (I’ve actually used it over an audio circuit between the BBC’s Delhi office and Bush House in London). It can be carried as a logic signal, as audio, over wire, optical fibre, or radio.

It’s not as fast as USB2, or even I2C. The AVR can manage RS232 data remarkably faster, certainly faster than you could decode biphase mark - but biphase mark is much more robust in terms of data rate, jitter, and changing clock rates.

In a production environment, you’d want to stiffen things up a little. You’d probably want to use an algorithm which compares the length of a previous cycle to decide whether a particular bit is a one or a zero, rather than using an absolute length. You’d probably want to provide some sort of packaging for the message – framing information, a more complex protocol, whatever. This is what you would do anyway in any comms method where you don’t have full control over both ends of the link, and the link itself.

Enjoy.

11 The source code

11.1 Makefile

```
biphase.o: biphase.c gcc -c biphase.c
unbiphase.o: unbiphase.c gcc -c unbiphase.c
biphase: biphase.o gcc -Wall -o biphase biphase.o
unbiphase: unbiphase.o gcc -Wall -o unbiphase unbiphase.o
```

11.2 biphase.c

```
// biphase.c
//
// (c) Nailed_Barnacle (Neil Barnes) 2007
//
// Released under LGPL conditions - see www.gnu.org for details.
//
// convert an input data file to an output .wav file, coding the data as
// biphase-mark data at a 1000bps data rate
//
// the signal is coded using RS232-type conventions; idle is high,
// a low start bit, eight data bits, and a high stop bit
//
// the bits themselves are coded such that there is a phase change
// at the end of every bit period, with a further phase change in
// the centre of the period in the case of a ‘1’. A zero (space)
// is therefore represented by a single high or low period of 1ms
// and a one (mark) by two 500us periods either low-high or high-low.
//
// the audio is coded onto a standard windows mono .wav file
// at a sample rate of sixteen ksps (it would work at any sample rate
// over about 8k, but 16k is convenient) and is sixteen bit; the
```
The signal used is 6dB down on peak amplitude so ranges from 0x4000 (high) to 0xc000 (low). The original signal is square wave but this may be modified by compression, noise addition, filtration etc.

// Note that the method of creating the waveheader structure implies 32-bit integers, 16-bit shorts, and 8-bit chars

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

// note that this is *not* the MS approved way to make RIFF headers
// they'd much rather you use their RIFF chunk management API which
// is, quite frankly, a pain in the rear

// And since I'm writing in linux, it's not immediately available
// This method produces legal .wav files without the headaches

// What we *should* have is RIFF chunk containing a WAVE chunk
// containing a 'fmt ' chunk and a 'data' chunk
// this will deliver it, providing we remember to fill in the missing bits later

struct _waveheader {
    char riff[4];
    int block_size;
    char wave[4];
    char fmt[4];
    int chunk_size;
    short tag;
    short channels;
    int samples_per_sec;
    int bytes_per_sec;
    short block_align;
    short bits_per_sample;
    char data[4];
    int data_size;
};

struct _waveheader wh; // header for wave files
FILE * fo; // file handles
FILE * fi;

#define SAMPLERATE 16000

// prototypes
int write_0 (FILE * file, int sample_rate, int last);
int write_1 (FILE * file, int sample_rate, int last);
int write_byte (FILE * file, int byte, int sample_rate, int last);

int main (int argc, char * argv[]) {
    static unsigned int count; // number of bytes in block
    int q;
    int r;
    short s; // the variable that holds the byte to be written to the .wav file
    int t;
    short last; // the value of the last phase of the signal
    char input; // byte read from the input file

    if (argc < 3) {
        printf("Please specify input and output filename...
        Usage: biphase infile outfile\n")
        exit(1);
    }
printf("Opening %s for generation\n", argv[2]);

// fill the header
// we don't yet know all the information
memcpy (wh.riff, "RIFF", 4); // there's always a RIFF to start
wh.block_size = 0; // we don't know how big the block will be, yet
memcpy (wh.wave, "WAVE", 4); // we have a WAVE chunk
memcpy (wh.fmt, "fmt ", 4); // which consists of a 'fmt ' chunk
wh.chunk_size = 16; // this big
wh.tag = 1; // PCM audio if tag = 1
wh.channels = 1; // mono
wh.samples_per_sec = SAMPLERATE; // 16k samples per second
wh.bytes_per_sec = 32000; // so obviously twice as many bytes per second
wh.block_align = 2; // the data is aligned on word boundaries
wh.bits_per_sample= 16; // and we will use sixteen bit audio so the mp3 codec won't barf
memcpy(wh.data, "data", 4); // then there's the 'data' block
wh.data_size = 0; // and all we know is that the data block is 36 bytes shorter
wh.block_size = 0; // than the RIFF block size

fi = fopen(argv[1], "rb"); // read binary as input
fo = fopen(argv[2], "wb"); // write binary as output
// some error checking here would be nice...
if ((fi == NULL) || (fo == NULL))
{
    printf("Can't open one of the input or output files... quiting\n");
    fcloseall();
    exit (1);
}

fwrite (&wh,sizeof(wh),1,fo); // write the header
// at this point, the header is written but the length bytes of the two critical chunks
// RIFF and 'data' are not yet known. We have to keep a track of how many bytes we write
// write a short burst of '1' as a idle start
last = 0xc000;
for (q = 0; q<50; q++)
{
    last = write_1(fo, SAMPLERATE, last);
    count += SAMPLERATE/1000;
}
// now we read in the filename and convert to audio on a byte by byte basis
while (fread(&input,1,1,fi) != NULL)
{
    last = write_byte(fo,input,SAMPLERATE,last);
    count += SAMPLERATE/100; // that's ten bit's worth
}

// and a few '1' s to idle at the end
for (q = 0; q<50; q++)
{
    last = write_1(fo, SAMPLERATE, last);
    count += SAMPLERATE/1000;
}
// tidy up the wave header
// we need to set the size of the RIFF block and the data block
// count has kept the number of samples written, we need bytes
count += 2;

// move to the appropriate place in the file
fseek(fo, 40, SEEK_SET); // the start of the data chunk size
fwrite (&count, sizeof(count), 1, fo);
count += 36; // increment count to accommodate the chunk headers following
fseek(fo, 4, SEEK_SET); // the start of the RIFF chunk size
fwrite (&count, sizeof(count), 1, fo);
printf("All done!\n");

12
fclose(fo);
}

int write_0 (FILE * file, int sample_rate, int last)
{
// write a half a squarewave to the audio output file
// number of samples written = samplerate/1000
// value of sample = inverse of 'last'
// returns value of sample written
short count;
short s;
int q;

    count = sample_rate/1000;  // to give a 1ms bit period
    if (last == 0x4000)
    {
        s = 0xc000;
        for (q = 0; q< count; q++)
            fwrite (&s,sizeof(s),1,fo);
        return 0xc000;
    }
    else
    {
        s = 0x4000;
        for (q = 0; q< count; q++)
            fwrite (&s,sizeof(s),1,fo);
        return 0x4000;
    }
}

int write_1 (FILE * file, int sample_rate, int last)
{
// write a squarewave to the audio output file
// number of samples written = samplerate/1000
// half these samples are high, half are low
// value of sample = inverse of 'last'
// returns value of sample written
short count;
short s;
int q;

    count = sample_rate/2000;  // to give a 500us half-bit period
    if (last == 0x4000)
    {
        s = 0xc000;
        for (q = 0; q< count; q++)
            fwrite (&s,sizeof(s),1,fo);
        s = 0x4000;
        for (q = 0; q< count; q++)
            fwrite (&s,sizeof(s),1,fo);
        return 0x4000;
    }
    else
    {
        s = 0x4000;
        for (q = 0; q< count; q++)
            fwrite (&s,sizeof(s),1,fo);
        s = 0xc000;
        for (q = 0; q< count; q++)
            fwrite (&s,sizeof(s),1,fo);
        return 0xc000;
    }
}

int write_byte (FILE * file, int byte, int sample_rate, int last)
{
// write a byte to the output file in the format
// start bit = 0
// data bit 0-7 low bit first
// stop bit = 0
// returns the last level written

int q;

// the start bit
last = write_0(fo, SAMPLERATE, last);

// the data bits
for (q=0; q<8; q++)
{
    if (byte & 1)
    {
        last = write_1(fo, SAMPLERATE, last);
    } else
    {
        last = write_0(fo, SAMPLERATE, last);
    }
    byte /= 2;
}

// the stop bit
last = write_1(fo, SAMPLERATE, last);
return last;

11.3 unbiphase.c

// biphase.c
//
// (c) Nailed_Barnacle (Neil Barnes) 2007
//
// Released under LGPL conditions - see www.gnu.org for details.
//
// convert an input .wav file containing a biphase-mark encoded audio
// file to an output binary file - no assumption is made about the
// actual data encoded.
//
// this program is built as a demonstration model only. In particular,
// it is deficient in verification of valid files, and in a real
// application a more robust packaging protocol would be used; possibly
// blocks protected with addresses, counts, and/or checksums.
//
// the signal is coded using RS232-type conventions; idle is high,
// a low start bit, eight data bits, and a high stop bit
//
// the bits themselves are coded such that there is a phase change
// at the end of every bit period, with a further phase change in
// the centre of the period in the case of a '1'. A zero (space)
// is therefore represented by a single high or low period of 1ms
// and a one (mark) by two 500us periods either low-high or high-low.
//
// the audio is coded onto a standard windows mono .wav file
// at a sample rate of sixteen kps (it would work at any sample rate
// over about 8k, but 16k is convenient) and is sixteen bit; the
// signal used is 6dB down on peak amplitude so ranges from 0x4000 (high)
// to 0xc000 (low). The original signal is square wave but this may be modified
// by compression, noise addition, filtration etc
//
// Note that the method of creating the _waveheader structure implies
// 32-bit integers, 16-bit shorts, and 8-bit chars
```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// note that this is *not* the MS approved way to read RIFF headers
// they'd much rather you use their RIFF chunk management API which
// is, quite frankly, a pain in the rear
//
// And since I'm writing in linux, it's not immediately available
// This method produces legal wav files without the headaches
//
// What we *should* have is RIFF chunk containing a WAVE chunk
// containing a 'fmt ' chunk and a 'data' chunk
// this will deliver it, providing we remember to fill in the missing bits later

struct _waveheader {
    char riff[4];
    int block_size;
    char wave[4];
    char fmt[4];
    int chunk_size;
    short tag;
    short channels;
    int samples_per_sec;
    int bytes_per_sec;
    short block_align;
    short bits_per_sample;
    char data[4];
    int data_size;
};

struct _waveheader wh; // header for wave files
FILE * fo; // file handles
FILE * fi;

// prototypes
int is_edge (FILE * file);
int get_period (FILE * file);
int get_bit (FILE * file);
int get_byte (FILE * file);

int main (int argc, char * argv[]) {
    static unsigned int count; // number of bytes in block
    int q;
    int r;
    short s; // the variable that holds the byte to be written to the .wav file
    int t;
    short last; // the value of the last phase of the signal
    char input; // byte read from the input file

    if (argc < 3) {
        printf("Please specify input and output filenames...
\n");
        printf("Usage: biphase infile outfile\n\n");
        exit(1);
    }

    printf("Opening %s for generation\n", argv[2]);

    // fill the header
    fi = fopen(argv[1], "rb"); // read binary as input
    fo = fopen(argv[2], "wb"); // write binary as output
    // some error checking here would be nice...
    if (((fi == NULL) || (fo == NULL))
```
printf("Can't open one of the input or output files... quitting\n.");
fcloseall();
exit (1);
}

// note - this is an *unsafe* way of reading RIFF files in general
// it assumes that the header will match *exactly* of the format
// defined by the _waveheader structure
//
// this is of course the case for the file we wrote in biphase, and
// with older files, but more modern files have more information in
// the header file and will not read correctly as some of the pointers
// will no longer align.
//
// so when you write your own routines, read the MS spec first

fread(&wh, sizeof(wh), 1, fi); // read the first eight bytes

// see if it's a RIFF file
if (strncmp("RIFF",wh.riff,4) != 0)
{
    // not a riff file
    printf("Sorry, that's not a riff file...\n");
    fcloseall();
    exit (1);
}

// good start, let's see if it's a WAVE file
if (strncmp("WAVE",wh.wave,4) != 0)
{
    // not a wave file
    printf("Sorry, that's not a wave file...\n");
    fcloseall();
    exit (1);
}

if (wh.channels != 1)
{
    // but it has to be mono...
    printf("Sorry, it must be a mono file...\n");
    fcloseall();
    exit (1);
}

// ok, that's enough checking for our purposes
// we'll assume then that the read pointer is pointing at the start of the data

int tmp = 0;
while ((tmp = get_byte(fi)) != -1000)
{
    fprintf(fo,"%c",(char)tmp);
    printf("All done!\n");
    fclose(fo);
}

int is_edge (FILE * file)
{
    // we return 0 if the sample we read from the audio file is the same phase
    // as the last sample we got (i.e. both were greater or lesser than zero)
    // return 1 if it was different and -1 if we ran out of file
    // our samples are all sixteen bits - short

    static short last_sample = 1;

    ...
short sample;

if ((fread(&sample, sizeof(short), 1, file)) != 0)
{
    if (((last_sample > 0) && (sample > 0)) || ((last_sample <= 0) && (sample <= 0)))
    {
        // no change in sign, therefore no edge
        last_sample = sample;
        return 0;
    }
    else
    {
        // signs different, therefore an edge
        last_sample = sample;
        return 1;
    }
}
else
{
    // ah, ran out of file...
    return -1;
}

int get_period (FILE * file)
{
    // get a single bit or half-bit - return the number of samples
    // to the next phase change
    // return -1 if we ran out of file

    int count = 0;
    int edge = 0;

    while (edge == 0)
    {
        edge = is_edge(file);
        ++count;
    }
    // ok, found an edge, check that we didn't run out of file
    if (edge == -1)
        return -1; // set the end flag
    else
        return count;

    int get_bit( FILE * file )
    {
        // recall that all bits have a transition at the end of each bit period;
        // '1's have a further transition in the centre of the period
        // this means that we can unambiguously decode 1 or 0 irrespective of
        // baud rate - a simple state machine will allow us to decode a single bit
        // and pass it back to a calling routine
        // in this particular case, we know our data is topped and tailed with a
        // short sequence of '1's, representing the idle state of the 232 data
        // if we *don't* know a starting state, we have to guess, and we can't
        // correct a long sequence of 1 or 0 until we get a bit change - this might
        // mean that there is data which is missed until synchronisation is
        // achieved, hence a need for rather better packeting/encapsulation than
        // is presented here
        // we return 1 or 0 if the bit is decoded, or -1 if we have a file error
        int period;
        static int first_half = 0; // this is set if we think we're in the
int middle;

// we read edges from the input file, and see how far apart they are
// since we know that the data rate is 1000 bits per second, and we have
// access to the bit rate, we can directly read the value of a bit:
// if the period is more than 750us then it's a zero; if it's less
// then it's half a one.
//
// a more generalised version can work at any speed by comparing
// the length of the current period with the length of the previous
// period and noting that you do not have an unambiguous result
// until you have observed a change from a 1 to 0 or from 0 to 1

period = 0;
middle = (wh.samples_per_sec*3)/(1000*4); // 3/4 of a bit period in samples

// get the period till the next edge
period = get_period(file);
// now see what to do with it
if (period == -1)
    return -1; // as we ran out of file
if (period > middle)
{
    // it's a zero
    first_half = 0;
    return 0;
}
else
{
    // it's half a one
    // if we've seen a half-one, then this next should be the second half
    // but if we're out of sync, it could be a zero
    // read the next edge and find out

    period = get_period(file);
    if (period == -1)
        return -1; // ran out of file again
    if (period > middle)
    {
        // ah, sync error, this is a zero then
        return 0;
    }
    else
    {
        // must be the second half of a one
        return 1;
    }
}

int get_byte (FILE * file)
{
    // return the single byte which has been received
    // return -100 if we run out of file
    // the data is presented as if it were an incoming rs232 stream -
    // high (1) idle, a low start bit, eight data bits, and a high stop bit

    int state = 0;
    int result = 0;
    int bit;
    int q;
// we're waiting for a start bit
while ((bit = get_bit(file)) == 1)
{
// wait and do nothing
}

if (bit == -1)
    return -1000; // ran out of file

// otherwise we got a start bit, probably, so get the rest of the word
// the data arrives bit 0 first
for (q=0; q<8; q++)
{
    bit = get_bit(file);
    if (bit == -1)
    return -1000;
    result |= (bit < < 8);
    result >>= 1;
}
// we could check that the next bit is a one - the stop bit - but
// instead we detect it while waiting for the start bit
return result;

11.4 biphase for AVR

    // biphase.c
    //
    // (c) neil barnes / nailed_barnacle@hotmail.com 2007
    //
    // released under LGPL - see www.gnu.org for details
    //
    // this is a demo program purely for the fun of it...
    // it reads an incoming audio signal with biphase mark (bpm) coded
    // data, as described in the AVRfreaks thread
    //
    // it requires a logic level bpm signal delivered to INT0/PD2
    // this can be generated by playing an audio signal from the PC
    // generated with the code on the AVRfreaks thread; convert it
    // to logic level using an opamp or comparator.
    //
    // the received bpm code - expected at 1kb/s - is output on TX0/PD0
    // as a serial data stream at 1kb/s. No data translation or verification
    // takes place.

#include <stdio.h>
#include <avr/io.h>
#include <string.h>
#include <avr/signal.h>
#include <avr/interrupt.h>

int main (void)
{
    // set the output port
    DDRD = 0x01; // all input except PD0

    // set the interrupt mask and mode
    MCUCR &= 0b11111100; // clear the INTL bits
    MCUCR |= 0b00000001; // any logic change cause interrupt
    GICR |= 0b01000000; // enable int0

    // set up a timer to give a useful count
    // we're expecting an interrupt every 4,000 to 8,000 counts
    // we can usefully clock timer 1 for that
    // the control bits are spread between TCCR1A and IB
TCCR1A = 0b00000000;  // normal counter, no output
TCCR1B = 0b00000001;  // normal counter, no prescale
sei();  // enable interrupts

while (1)
{
  // if you have anything useful to do, here is where to do it
  return 0;  // just kidding...
}

SIGNAL (SIG_INTERRUPT0)
{
  // we arrive here in the vain hope of having received an
  // interrupt from INT0/PD2

  int period;  // somewhere to keep the period
  static int first = 0;  // has the first half been received?

  // we need to know how long it was since the last interrupt
  period = TCNT1;  // this grabs all sixteen bits
  TCNT1 = 0;  // so clear the count for the next bit

  if (period > 6000)
  {
    // definitely a zero...
    PORTD &= 0b11111110;  // so clear the output
    first = 0;
    return;
  }
  else
  {
    // hmm, a zero, but which half?
    if (first == 1)
    {
      // then it's the second half
      PORTD |= 0b00000001;  // set the output
      first = 0;
      return;
    }
    else
    {
      // it's just the first half
      first = 1;
    }
  }
}