ECE 380

Mini-Project: Frequency-Shift Keying Receiver

Introduction

The goal of this lab is to understand a simple modem, the Frequency Shift Keying (FSK) Modem, referred to by the International Telecommunications Union (I.T.U.) as V.21. Here is a quick recap of the operation of the v.21 FSK modem. The V.21 modem communicates 1's and 0's by sending either a 1650 Hz tone or a 1850 Hz tone, respectively, for 1/300 sec. Thus the overall data rate is 300 bits/second (one bit is sent in 1/300-th of a second). Even though 300 bps is quite slow in comparison to the theoretical maximum of 56 kilobits per second over a phone line, the V.21 protocol is still used in almost every modem call, because receiving it is so simple. A V.21 modem call can be received without using difficult techniques such as equalizers, cancellers and matched filters. Furthermore, it can be received accurately even in the presence of a significant amount of noise. For these reasons, V.21 is used to perform the initial handshake between two modems, meaning that V.21 is a way to communicate some basic startup and control information between the two modems. You can hear the V.21 modem tones at home when your V34, V.90, V.92 phone line modem or fax machine starts a phone call. V.21 is also used to transmit caller ID information over the phone line.

Modulator

In the previous miniproject you built a FSK modulator for encoding text data into binary data and then into a FSK waveform. You can use your modulator to generate signals for this lab.

Demodulator

The receiver for V.21 must determine which of the two tones is present, and must make this decision every 1/300-th of a second. Furthermore, the receiver must synchronize with the bit interval, meaning that it must learn where the starting and ending times of each bit are located. This synchronization is essential to making reliable "0-1" decisions because the transition times must be avoided. A block diagram of the FSK V.21 demodulator is given in Fig. 1. Each of the main sections will be described in more detail below.

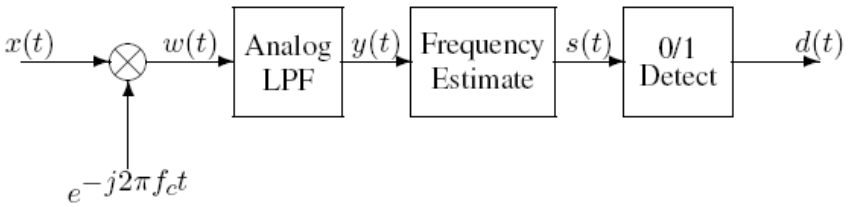


Figure 1: Block diagram of the FSK V.21 demodulator.

Mixing

A basic operation that most modems need to perform is frequency shifting of the input signal. According to the frequency-shifting property of the Fourier Transform, this can be done by simply multiplying the input signal by a complex exponential.

$$w(t) = x(t)e^{-j2\pi f_c t}$$

This effect can best be understood by thinking of *x(t)* as a sum of complex exponentials and observing what happens to each individual frequency component. In this case, x(t) is

$$x(t) = \cos(2\pi f_1 t) = \tfrac{1}{2}e^{j2\pi f_1 t} + \tfrac{1}{2}e^{-j2\pi f_1 t}.$$

When *x(t)* is multiplied by the given complex exponential at frequency $-f_c$ Hz, the exponents simply add
and the resulting *w(t)* is as follows:

$$w(t) = \tfrac{1}{2}e^{j2\pi(f_1 - f_c)t} + \tfrac{1}{2}e^{j2\pi(-f_1 - f_c)t}.$$

Note that the new frequencies in *w(t)* are simply the old frequencies shifted down (to the left on the *f* axis) by f_c, i.e., $f_1 - f_c$ and $-f_1 - f_c$. Also note that *w(t)* is complex so we no longer have the condition of complex conjugate symmetry between the two complex exponential components. This simply means that now our signal is really two signals: a real part signal and an imaginary part signal. If we want to filter this complex signal with a real filter, we simply filter the real part and the imaginary part separately with the same filter. In LabVIEW, the FIR filter VI does this very thing for you. The purpose of the filter is to remove the complex exponential with frequency $-f_1 - f_c$ while leaving the other component whose frequency is $f_1 - f_c$. Thus, the filter output should be of the form

$$y(t) = Ae^{j2\pi(f_1 - f_c)t},$$

where *A* will depend on the gain of the filter in its passband. For V.21 FSK, we will choose f_c so that the original frequencies of +1650 Hz and +1850 Hz in *x(t)* are shifted to −100 Hz and +100 Hz respectively, and these are the frequencies passed by the filter. This is achieved by choosing $f_c = 1750$.

Discrete-Time Simulation

The MATLAB implementation of the FSK V.21 modem is a simulation. This means that even though we seem to have continuous-time signals such as x(t) and w(t), we actually use sampled versions in the MATLAB program. For this simulation, we will choose the input sampling frequency to be

$$\boxed{f_{\text{samp}} = 9000 \text{ Hz.}}\,[1]$$

This value is chosen to be rather high so that the input signal will appear to be continuous in plots. In Fig. 2 the continuous-time signals have been replaced with their sampled versions, e.g., *x(t)* becomes *x[n]*, *w(t)* becomes *w[n]*, and so on. The complex exponential used in the mixer must also be converted from to a discrete-time signal $e^{-j\hat{\omega}_c n}$. What is $\hat{\omega}_c$?

[1] You will have to modify your transmitter to work at f_{samp} = 9000 Hz. This is done to avoid problems with noninteger numbers of samples per bit. An "extra" you could add would be to make your transmitter and receiver work with noninteger numbers of samples per bit, such as 300 bps at f_{samp} = 8000 Hz.

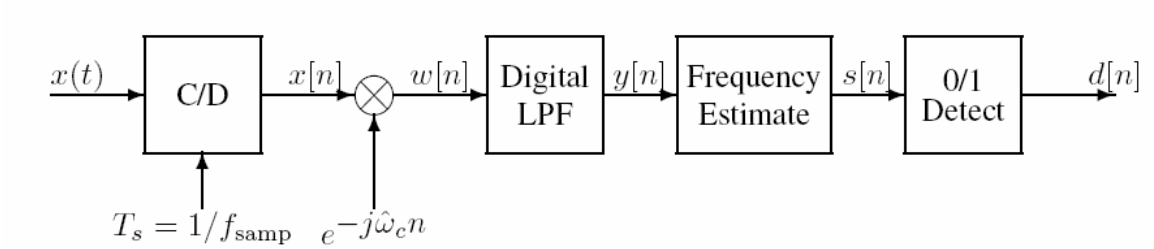


Figure 2: FSK system demodulator system simulated as a discrete-time system at a sampling rate of f_{samp}.

Low Pass Filter Design

The other thing that must be done in converting the block diagram of Fig. 1 to the simulation block diagram in Fig. 2 is to replace the analog filter by a digital filter. This filter must pass the band of frequencies equivalent to ±100 Hz, and it must remove the higher frequencies generated at the output of the mixer which are $-f_c - 1650 = -3400$ and $-f_c - 1850 = -3600$ Hz. You should already be familiar with some methods for designing FIR low pass filters with given specifications on the passband and stopband edges, as well as constraints on the ripple characteristics in the pass and stop bands. Such designs can be carried out in LabVIEW with the Digital Filter Design GUI, or with the filter design VIs. Since the digital filter is typically used to filter signals that are sampled analog signals, it is important to know how the bandedges of the digital filter can be expressed in terms of the desired analog cutoff frequencies. This can be done if the sampling frequency f_{samp} is known. For example, if we want to have a lowpass filter with an effective cutoff frequency of 2000 Hz, and we are using a digital filter running at fsamp = 8000 Hz, then the digital filter must have its cutoff frequency[2] at $\hat{\omega} = 2\pi(2000/8000) = \frac{1}{2}\pi$. The suggested design programs handle this for you.

In the FSK V.21 system, the frequency shifting of the mixer will generate spectrum lines that must be removed by filtering. If we demand that these unwanted spectrum components must be reduced in magnitude by a factor of 100, then we have given the specifications on the stopband ripple. A reduction by a factor of 100 means that the stopband ripple must be less than 0.01, or −40 dB. The passband, on the other hand, must be made wide enough so that the desired frequency components will go through the LPF with little or no change. Since the LPF's frequency response will have a passband ripple, we will use a specification on the passband of 1 dB, which forces the passband magnitude to lie between 0.89 and 1.12.

[2] Remember that the frequency response of a digital filter, $H(e^{j\hat{\omega}})$, is a function of the frequency variable $\hat{\omega}$ that runs from $\hat{\omega} = -\pi$ to $\hat{\omega} = +\pi$.

Slicing

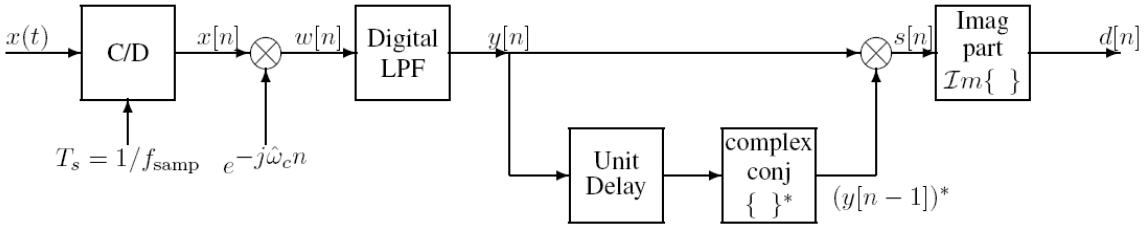


Figure 3: Frequency estimation in a dual-frequency FSK system can be performed with a slicer.

Another basic operation of most modems is to measure the frequency of a received tone. This could be accomplished by optimal filtering algorithms such as matched filters designed to enhance the tones of interest. However, if you are guaranteed to be looking at only one tone at a given time and the noise is not severe (both of which are true for V.21), then there is a simpler method that can be employed to save computation when measuring frequency.

Slicing is defined as follows:

$$s[n] = y[n](y[n-1])^*.$$

It is a non-linear operation, but in the case where *x[n]* is a single cosine input, the filter output will be of the form $y[n] = Ae^{j\hat{\omega}_0 n}$, where $\hat{\omega}_0 = \mp 200\pi / f_{samp}$. In this case, the output of the slicer reduces to

$$s[n] = (Ae^{j\hat{\omega}_0 n})(A^* e^{-j\hat{\omega}_0 (n-1)}).$$

After adding the exponents, the output simplifies to

$$s[n] = |A|^2 e^{j\hat{\omega}_0}.$$

If the objective were to determine the frequency $\hat{\omega}_0$, then it is sufficient to take the imaginary part

$$d[n] = \Im m\{s[n]\} = \Im m\{|A|^2 e^{j\hat{\omega}_0}\} \qquad \Longrightarrow d[n] = |A|^2 \sin(\hat{\omega}_0)$$

and use *d[n]* to calculate the ArcSin(·) to get an estimate of $\hat{\omega}_0$. However, the FSK V.21 system is even simpler than that, because we only need to decode two cases: a zero or a one. When $\hat{\omega}_0 < 0$ we have a "1", and when $\hat{\omega}_0 > 0$ we have a "0." In addition, the sign of $\hat{\omega}_0$ is the same as the sign of $|A|^2 \sin(\hat{\omega}_0)$, so we only need to check the sign bit of *d[n]* to perform the decoding. As will be seen in the final implementation of this lab, the recovery of the V.21 signal reduces to discriminating between a +100 Hz tone and a −100 Hz tone. Taking the imaginary part of *s[n]*, the slicer output, will provide an easy way to determine whether a 1650 Hz or 1850 Hz tone was originally present. Thus we can define *b[n]* as an estimate of the bit that is represented by the slicer output at time *n*.

$$b[n] = \begin{cases} 0 & \text{when } d[n] \geq 0 \\ 1 & \text{when } d[n] < 0 \end{cases}$$

In LabVIEW, you can use the sign VI to implement (3) by implementing the function (either in VIs or as a formula) $b[n] = (sign(-d[n]) + 1)/2$.

f_{original}	f_{mixed}	$\lvert A\rvert^2 \sin \hat{\omega}_0$	Bit Received
1650 Hz	−100 Hz	Negative	1
1850 Hz	100 Hz	Positive	0

Table 1: FSK Decoding Rule

Decoding the Bits

Now we must extract the bit information from the output signal *d[n]*. To get a better understanding of this problem, let us first describe the input signal to our demodulator system. Remember that the sampled waveform *x[n]* is a variable frequency sinusoid that can switch frequency every 30 samples (assuming bit rate of 300 bps and sampling rate of 9000 samples/sec) and the sinusoid switches between 1650 Hz to represent a 1 and 1850 Hz to represent a 0. First of all, in any realistic simulation, the first part of the signal should be zero (or very small if there is noise); i.e., this part of the signal represents the time before the modem starts sending data. Then we will have a segment of the waveform that encodes a "preamble" and "marker flag" that will allow us to synchronize on the bits as well as the bytes. This part of the waveform will encode the bit string [110011001100110011111111]. The next part of the waveform will encode the bit sequence of the message data. Finally, the last part of the waveform encodes the "end marker", which is the bit string [1111111111111111]. This is summarized in the following table:

Sync Pattern	Start Flag	DATA	End Flag
`1100110011001100`	`11111111`	`message bits`	`1111111111111111`

In our discrete-time simulation, the signal *d[n]* at the output of the slicer is either positive or negative at each sample time, and we have just seen that the sign of this signal is an indicator of whether a 1 or a 0 is being encoded at any particular sample time n. Now we should remember that if the bit rate is 300 bps and the sampling rate is 9000 Hz, this means that each bit of the encoded message is actually represented by a group of 30 consecutive samples. Since the bits are transmitted at a uniform rate, this means that we only need to examine a sample of *d[n]* every 30 samples to determine the sequence of bits that is encoded in the signal *x[n]* (or equivalently *d[n]*). The problem is to find out when the signal represents data and then to synchronize our sampling of the *b[n]* sequence so that we can not only find out what the bits are, but also be able to group the bits into 8 bit bytes so that we can decode the bit stream into ASCII characters. Basically, there are three steps: (1) finding the preamble that marks the beginning of the message, (2) synchronizing with the bit interval, and (3) grouping the bits into 8-bit sets to be converted into ASCII characters.

Bit Synchronization

The key to synchronization is that we have a known pattern of bits at the beginning of the message. We must take advantage of this to determine where the bits change state. The preamble and marker pattern [110011001100110011111111] begins after a period of silence (or noise). Thus we can expect an abrupt change at the beginning of this pattern and then a sequence of transitions from 1 to 0 and then from 0 to 1. If we do not count a transition at the beginning, we can expect 8 transtions between the beginning of the preamble and including the beginning of the marker bits [11111111]. We can use this

knowledge of the signal structure to locate this pattern and then use this to anchor our search for the message bits. Thus, we want to begin by locating the transition points. A simple indicator of this would be

$$p[n] = |b[n] - b[n-1]|.$$

This sequence should be +1 at each transition point. For noiseless signals in our simulation, these indicator points should be exactly 60 samples apart in the preamble part because it consists of four groups of two consecutive ones followed by two consecutive zeros, etc. In a practical situation, there may be some "jitter" in the locations that you might need to accommodate. In any case, if we locate the beginning of the signal, then after about 8 × 60 = 480 samples, we should find the eighth marker, which is the beginning of the group of eight 1s in the marker.

Byte Synchronization

The purpose of the marker section is to provide more reliability in grouping the bits into groups of 8 for ultimate decoding into ASCII symbols. Since there are eight 1s, the number of samples of *b[n]* between the last transition of the preamble (first transition of the marker) and the first transition of the message bits should be 8 × 30 = 240 samples in our simulation since at 300 bps and a sampling rate of 9000 Hz, each individual bit spans 30 samples. Once the end of the group of marker bits has been found, we have achieved byte synchronization since from that point on, groups of 240 samples of *b[n]* are known to represent the eight bits of one ASCII symbol.

Determining the Bit Sequence and forming Bytes

After we have found the end of the marker, we can determine the sequence of bits by simply looking at the signal during each bit interval. Since each of these intervals is 30 samples long, we can find the middle (where the answer is likely to be most robust) by simply "sampling" the sequence *b[n]* at a point that is offset by 15 samples from the beginning of each bit interval. Since the end of the marker interval is the beginning of the message bits, we can use as our estimate

$$f[k] = b[n_{\text{beg}} + 15 + 30k],$$

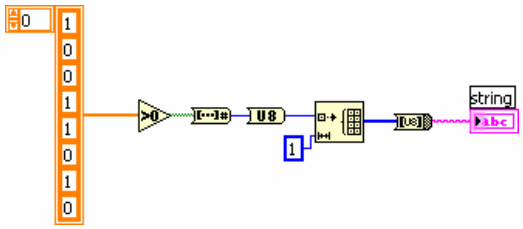
where n_{beg} is the sample location of the end of the marker bits (beginning of the message bits). The end of the message is marked by 16 consecutive 1s, which decode to 255, 255. Finally, after we have determined the sequence of bits and grouped them into bytes, we must convert the resulting binary number into ASCII symbols. This completes the decoding simulation. How to do this in the LabVIEW simulation is suggested below.

Components

The easiest way to build up a receiver is to build the component functions separately, then combine them to form the entire system. In this section we will build up some of the component functions necessary for a full receiver.

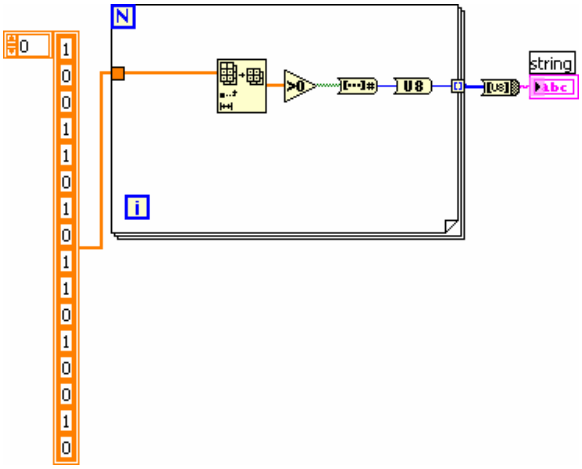
Binary Stream to ASCII

The FSK decoder involves the operation of converting 8-bit patterns to ASCII. The LabVIEW VIs shown can convert a bit stream to ASCII. Try the following example:



There are several stages of conversions in this example because we must convert to the correct datatype for the subsequent conversions. The need to convert the byte into a single element array won't be necessary when we have a list of bytes to convert. In order to understand this example, you might want look at the context help for each of the VIs shown. They are "Greater Than Zero?", "Boolean Array to Number," "To Unsigned Byte Integer," "Initialize Array," and "Byte Array To String." You should probe the values to see the decimal value that matches the character. Is the bit array being interpreted LSB first or MSB first?

Modify the example above so that it can do more than one ASCII character. Complete the loop below:



Check the byte values given to determine the ASCII values that should result. Note that the N input to the loop must be wired, and so must the index and length inputs on the Array Subset VI. Finally, it is important that you disable indexing on the loop's array input and enable indexing on the byte output.

Mixing Moves the Spectral Lines

The input signal to the mixer is a real sinusoid, so it has spectral lines at either ±1650 Hz, or ±1850 Hz. The mixer multiplies by a complex exponential, $e^{-j2\pi(1750)t}$. For both input cases, determine the location of all the spectral lines after mixing. Recall from the Fourier transform properties the effect of multiplying by a complex exponential.

LPF Design

After mixing, the signal of interest is either +100 Hz or −100 Hz. Since the sampling rate is f_{samp} = 9000 Hz, the desired passband of the LPF can be calculated in the $\hat{\omega}$ domain: call the result $\hat{\omega}_p$. The stopband of the digital LPF must be chosen to remove the extra spectral lines at the output of the mixer. Determine the stopband cutoff frequency that will be needed to remove these extra spectral lines. Design the digital FIR lowpass filter

that will have these bandedges. Make the stopband ripple less than −40 dB, and the passband ripple less than 1 dB. The FIR filter that will meet these specs is rather short — determine its length, i.e., number of filter coefficients. If you use either the Digital Filter Design GUI or the FIR Windowed Filter VI, you can work directly with the analog frequencies, once you enter f_{samp} = 9000 Hz into the GUI.

Test the Mixer

In this section you will need to use your encoder to generate test signals. First test the encoder with the mixer and the lowpass filter to be sure that the output of the filter is a complex exponential with either +100 Hz or -100 Hz frequency.

(a) You must use the filter design specified in the previous section.

(b) Use your encoder with bps = 300, and f_{samp} = 9000 to create an FSK signal for a bit stream. You can make the string sent something simple like 'Test.'

(c) Process the FSK signal created in the previous part through the mixer (with f_c = 1750 Hz) and the lowpass filter that you designed above. Make a plot of the real part of the output signal in the time domain so you can see how the bits change.

(d) It is possible to view the spectrogram of the input and output of the lowpass filter, but some care is needed. First of all, for the 300 bps signal, an extremely short window length is needed, e.g. 30, because the bit duration is only 30 samples at f_{samp} = 9000 Hz. You can set the window length at the bottom of the spectrogram tab in the TripleDisplay. You also need to set the time increment to 1. A complex spectrogram VI has been provided on Angel. It converts the complex signal into a real signal whose spectrum can be displayed in a TripleDisplay. Create an indicator from its output, and set the spectrogram settings there. The complex input signal extends from 0 to f_{samp}, so negative frequency components actually show up at high frequencies. For example, −2000 Hz would show up at 9000 − 2000 = 7000 Hz when f_{samp} = 9000 Hz.

Slicer Implementation

The objective of this section is to implement the slicer and show that it gives a constant output when the input is a single-frequency complex exponential.

(a) For the input to the slicer, use the filtered output signal from the previous section. Or, if you are unsure of that output, make a test signal by synthesizing *y[n]* as a single complex exponential:

$$y[n] = e^{j\hat{\omega}_1 n}$$

Find the correct value for $\hat{\omega}_1$ from fs = 9000 Hz and f_1 = 100 Hz.

(b) Note that *y[n − 1]* can easily be obtained from *y[n]* by with a delay of one sample. Recall that the filter VI will keep the vectors the same size, and the correct FIR filter coefficients used to create *y[n − 1]* are easy to determine:

(c) Now *s[n]* (the slicer output) can be obtained by using the Complex Conjugate VI in LabVIEW. Plot the imaginary part of the slicer output and compare this to the predicted value which is a constant:

$$\sin(2\pi(100)/9000)$$

Putting the Pieces Together

In the Components section, you should have completed the implementation of all parts of the FSK demodulator, so the only thing left is the "decoder" that will synchronize on the bits and group them into 8-bit bytes.

Transmitter/Modulator

You should use your own FSK transmitter to generate the test signals you need. The transmitter/modulator must be set to run at a sampling rate of 9000 Hz. A realistic simulation would use 8000 Hz because phone lines are 8000 Hz, but 9000 Hz keeps our sampling rate an integer multiple of 300 which is our symbol/bit rate. One useful test message is '@U ' because it contains runs of consecutive zeros, consecutive ones, and also alternating zeros and ones. As an "extra" you could add an optional input parameter to your transmitter that will allow you to add noise to the FSK signal for a more realistic simulation. It is probably a good idea to begin your program development and initial testing with no noise.

Synchronize on the Bits

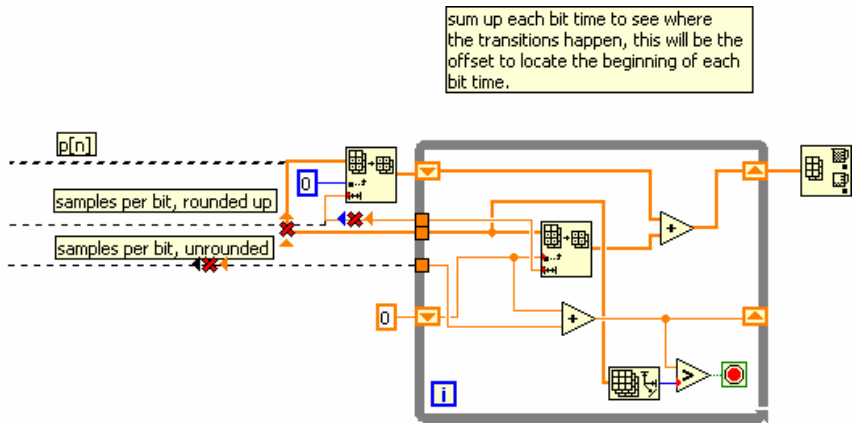
You should extend your simulation that you developed in the Components Section to include the computation of the signal *b[n]* as given above. Use the FSK Modulator to produce a test signal for a very short message, and run that signal through a program consisting of the mixer, LPF and slicer to produce the output *b[n]* which are the detected zeros and ones obtained from *d[n]*. For testing the synchronization algorithm that you will write, you will concentrate on the front end of *b[n]* where the preamble is found. Since *b[n]* is sampled at f_{samp} = 9000 Hz, each bit interval lasts for 30 samples and each pair of ones and each pair of zeros in the preamble lasts for 60 samples. The goal here is to find the transitions (or jumps) in *b[n]*, which also correspond to sign changes in *d[n]*. This could be accomplished with a for loop. A good starting point is to compute the transitions in *b[n]* using the "transition" signal *p[n]*. Note that *p[n]* can be computed using a simple FIR filter and an absolute value. Ideally these transitions should be exactly 60 samples apart in the preamble region of *p[n]*.

Code for Synchronization

The goal of bit synchronization is to reduce the many samples per bit present in *b[n]* down to just a single sample per bit, representing the originally sent bit sequence. To do this, we must identify a sample near the middle of each bit. Since the bps of the transmitted signal determines how far apart our sample times should be (30 samples per bit in this example), we only need to know a relative shift from the beginning of the file to the center of a bit. This quantity is unknown if we don't know how much silence could be present before the transmitter turns on.

With *p[n]*, you can identify the beginning of each bit transition, and each one should be a multiple of one bit-time apart. Since we know the bps of the transmitter, we only need to know how many samples to skip to get to the middle of a bit time. A simple way to accomplish this is to divide the sequence *p[n]* into segments one bit-time long, and add them up. If each segment begins one bit-time later than the last (tricky if the samples per bit isn't an integer), then the transitions will all happen at about the same shift and index

of the largest value of the sum will be the correct number of samples to skip to find the beginning of each bit-time in the file. Of course, we want to skip to the middle of each bit, not to the beginning.



The code above uses a while loop and sums one bit-length chunks of *p[n]* until the end of the signal. It is enough to find just a few transitions, but you must be sure to run long enough to get past any silence before the preamble. In a real streaming system, this bit synchronization would need to be adjusted regularly, since two independent clocks are never exactly the same frequency, but for a short message, we can just synchronize once.

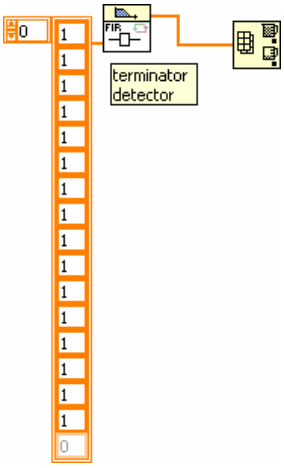
Add your own code to sample values from *b[n]* in the middle of each bit-time. At this point, you may be including invalid bits from silence, the preamble bits, the message bits, and the terminator bits. We will locate the preamble and terminator sequences in the next section.

Get the Message Text

The final step of the decoder is to turn the bit patterns back into text. Consult the Binary Stream to ASCII Section above for the code needed to do this efficiently. Recall that the FSK transmitted data is organized as follows:

Sync Pattern	Start Flag	DATA	End Flag
1100110011001100	11111111	message bits	1111111111111111

One way to locate the preamble and terminator is using an FIR filter directly on the bitstream data. When convolving a time reversed copy (*pattern[-n]*) of the desired pattern with the bitstream, the output will be maximized when all the data bits match the pattern bits. The index of the maximum value will locate the desired pattern in the data stream, and the data between the preamble and the terminator will be the desired data message. In the code example shown below, the time reversal is not apparent due to symmetry.



Three test messages are supplied in the FSK Receiver Test Data VI. For the first one, only the data bits are given as 1's and 0's; the second one contains all of *b[n]* including the preamble. The *b[n]* values are given as 1's and -1's instead of 1's and 0's. It is easy to convert them if you choose to. You should be able to verify that you have correctly recovered the data for both of these. You can use the first test to check that your "bits to ASCII" code is working. The second one will allow you to test your code for detecting the preamble and terminator and extracting the message data. Then you are ready to run the entire FSK demodulator/decoder to recover a message from the last test message which is an FSK signal having the same data format as shown above, but this time you will have to demodulate the signal to get the data unlike the first two test messages that gave you the transmitted data bits. This is the waveform constant. Please note that the DATA portion of each message is actually text, so you should get something that is readable as a sentence in English. If you correctly group the received data into bytes (8 bits), and convert each byte into its ASCII character you will be able to read the messages.

The test messages are provided at both the f_{samp} = 8000 Hz rate and the f_{samp} = 9000 Hz rate.

What's Due

This miniproject is optional. You will get extra credit for completing it well.

For this mini-project you are to work on your own. You may discuss your ideas with others, but you may not share LabVIEW files. The policy stated on the Homework Procedures handout under "Responsibility" applies to this and other mini-projects.

What is due:

1. A brief memo describing what you did. Be sure to have a sentence or two intro and conclusion.
2. Plots of the specgram or sketches of the spectrum for the signals into the mixer, out of the mixer, and after the LPF.

3. A plot of *b[n]* showing where the decisions will be made, based on the synchronization that found the middle of the bit interval.
4. A description of your technique for synchronizing the bits and bytes, based on your knowledge of the training signal and the data flags.
5. A comparison of the received bits to the transmitted pattern for a simple case. These ought to be identical when there were no distortions, which is the (unrealistic) condition for this miniproject.
6. Print out your LabVIEW code. Selecting File→Print will start a print wizard. When you reach the "Print Contents" window, select "Icon, description, panel, and diagram." This will print out the whole works in a compact format.

A hard copy of the memo and LabVIEW printout are to be handed in at the start of class on **Thursday, May 24**.

Supplementary Thoughts

In this lab we built a modem, a digital form of communication, which is generally speaking far superior to analog communication. The magic of digital is that if the bits are recovered correctly (and there are a variety of advanced techniques to achieve this), then all the distortions of the channel, noise, and modem imperfections along the way are effectively eliminated. In this lab, the digital magic occurs when the decision is made to convert the slicer output *d[n]* into *b[n]* which is either a 0 or a 1 bit. A cell phone is a good example of the magic of digital. In spite of the horrible channel between you and the cell tower (which is changing as you cruise down the interstate while talking on your cell phones) and all the ambient noise, the data bits can be heavily encoded and recovered properly, thus achieving digital quality.

The system described in this miniproject handout is adequate to decode a low noise, low distortion signal captured as a single array of sample data. However, real systems often must detect whether the transmitter is on or not, and run on continuous streams of data, in noisy and distorted environments. With further modification, it would be possible to convert the system you have already created into an "air chat" device that sent text messages over the speakers of your computer to another computer where the system would have to continuously "listen" to the microphone input to watch for the preamble, then display a message whenever one was detected. The received signal would be much noisier and more distorted after passing through a real channel, but should still be workable at low bitrates and short distances. If you try this, you should be aware that the microphones on some laptops are limited to fairly low frequency inputs, so check your input signal carefully and do not assume your laptop is "hearing" what you hear.